课程内容

日期	知识模块	知识点
2月20日	课程介绍	Linux 内核基本结构、Linux的历史、开源基础知识简介、驱动程序介绍
2月27日	实验课一	内核编译，内核补丁
3月5日	内核编程基础知识概述	内核调试技术、模块编程、源代码阅读工具、linux启动过程、git简介
3月12日	实验课2	内核调试
3月19日 3月26日	进程管理与调度	Linux进程基本概念、进程的生命周期、进程上下文切换、Linux 进程调度策略、调度算法、调度相关的调用
4月2日	实验课3	提取进程信息
4月9日	系统调用、中断处理	系统调用内核支持机制、系统调用实现、Linux中断处理、下半部
4月16日	实验课4	添加系统调用、显示系统缺页次数
4月23日	内核同步	原子操作、自旋锁、RCU、内存屏障等linux内核同步机制
4月30日	内存管理1	内存寻址、Linux物理内存和虚拟内存的组织、伙伴系统、vmalloc
5月14日	内存管理2	Slab分配器、进程地址空间
5月21日	实验课5	观察内存映射、逻辑地址与物理地址的对应
5月28日	文件系统	Linux虚拟文件系统、Ext2/Ext3/Ext4文件系统结构与特性
6月4日	Linux设备驱动基础字符设备驱动程序设计	Linux设备驱动基础、字符设备创建和加载、字符设备的操作、对字符设备进行poll 和select的实现、字符设备访问控制、IOCTL、阻塞IO、异步事件等
	基于linux的容器平台技术概述+实验课6	虚拟化技术与容器、Docker概述、Kubernetes概述 实验：Docker对容器的资源限制
6月11日	报告课	期末课程报告

Linux Filesystem

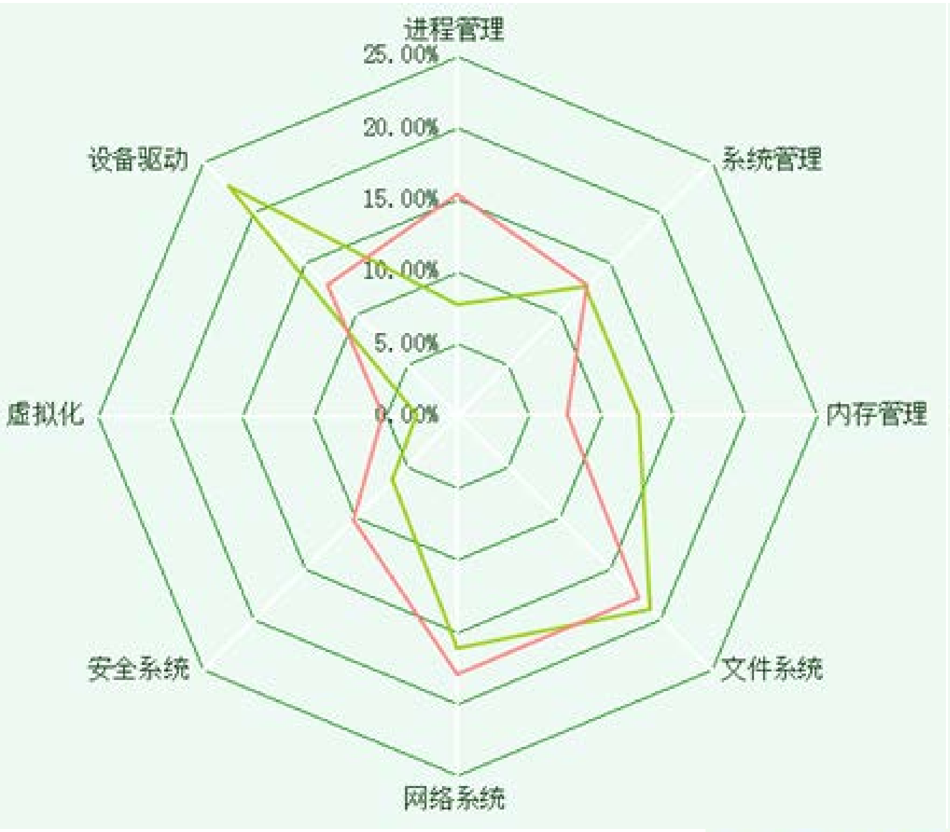
荆琦

jingqi@pku.edu.cn

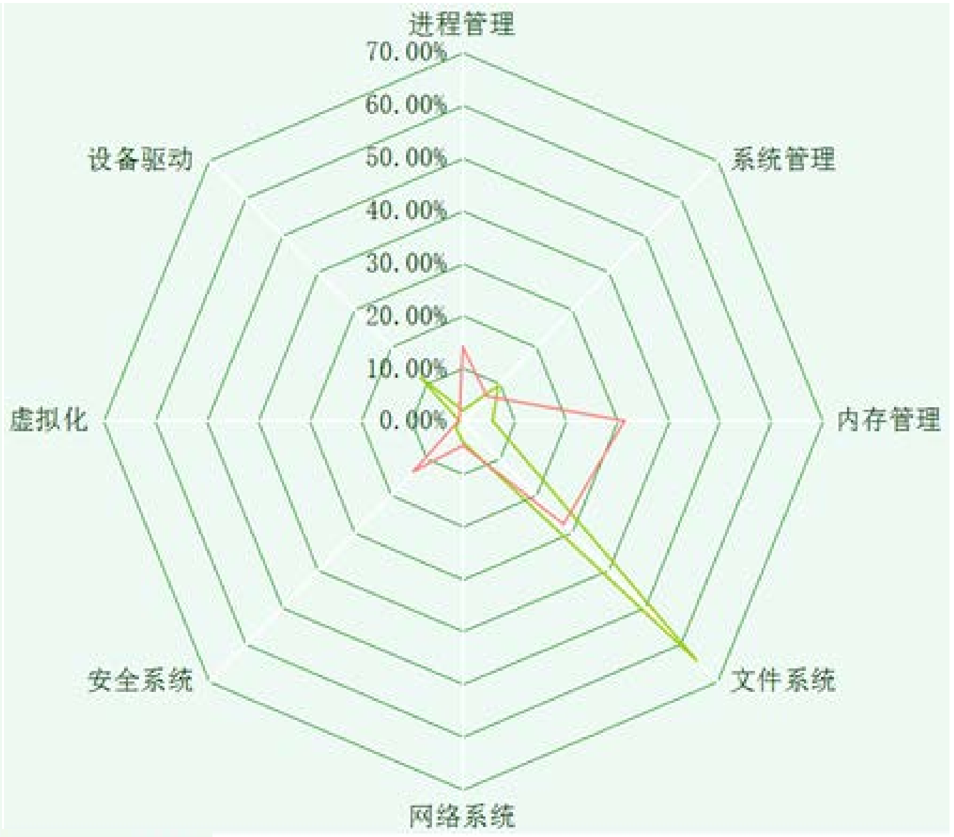
北京大学软件与微电子学院

Linux 3.5到3.19版本与此间longterm版本各子系统特性数据对比

Linux 3.5到3.19和longterm版本变更**特性数**占总特性数比例对比图



Linux 3.5到3.19和longterm版本变更**代码行数**占总变更代码行数比例对比图



🟩 Linux 3.5-3.19
🟥 Linux longterm

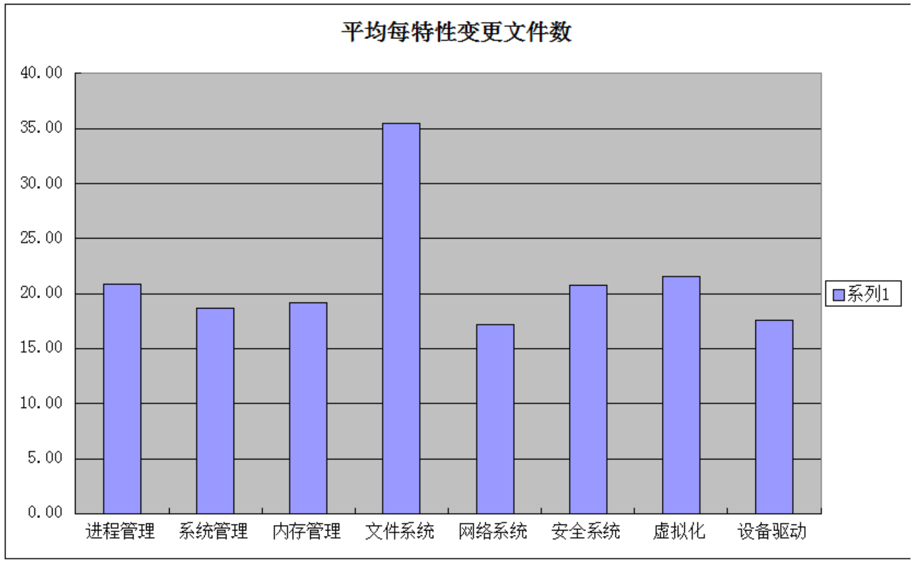
重要特性分析

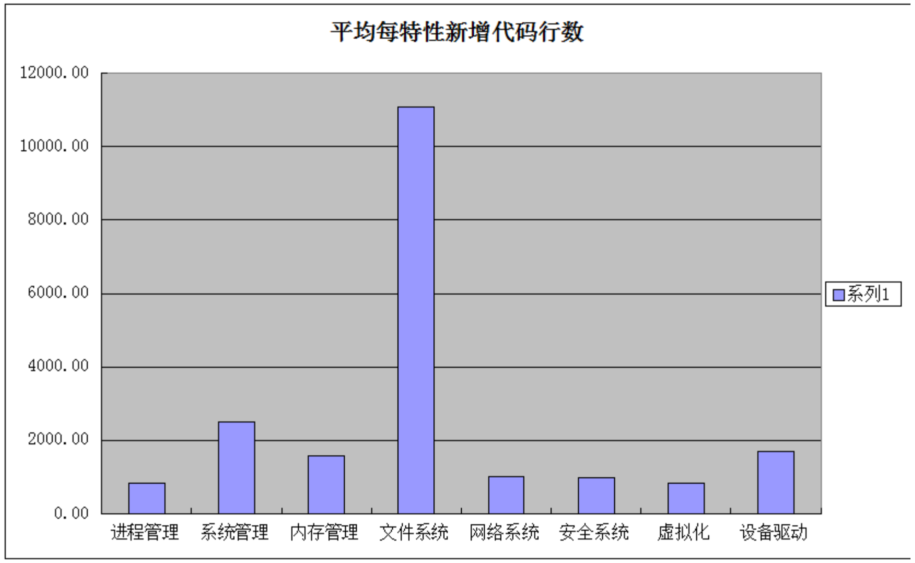
从下图是每特性变更代码量和文件数的统计，从该角度来看各子系统特性分析难度：

●文件系统分析难度最大，因为平均每特性涉及的文件数最多、修改的代码量也最大。

●进程管理、安全系统和虚拟化变更涉及文件较多，但代码分析工作量相对较小。

●系统管理、内存管理、网络和设备驱动子系统变更涉及的文件数较少，但代码工作量相对较大。



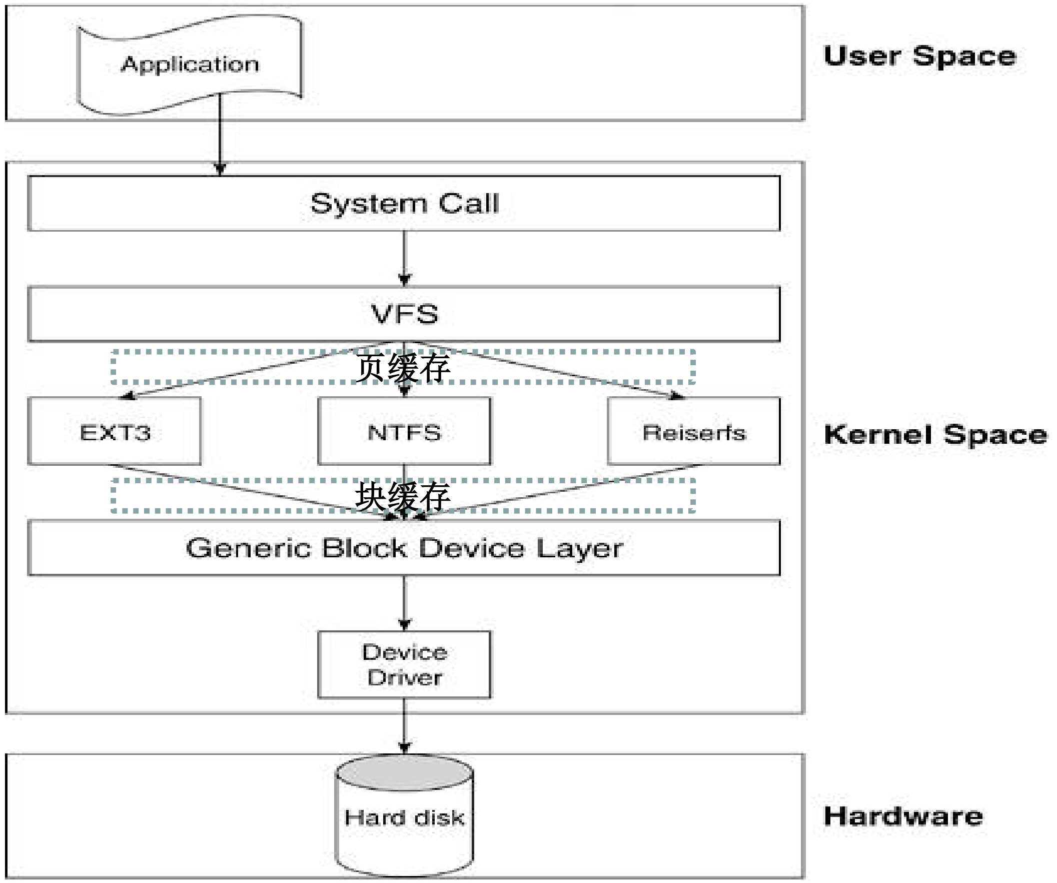


内容

- Overview of Linux Virtual Filesystem
- Ext2 文件系统 (Second Extended Filesystem)
- Ext3 & Ext4

- Part 1 Overview of Linux Virtual Filesystem

Linux VFS



- The user application accesses the generic VFS through system calls
- Each supported filesystem must have an **<u>implementation of a set of functions that perform the VFS-supported operations</u>** (for example, open, read, write, and close).
- The VFS keeps track of the filesystems it supports and the functions that perform each of the operations

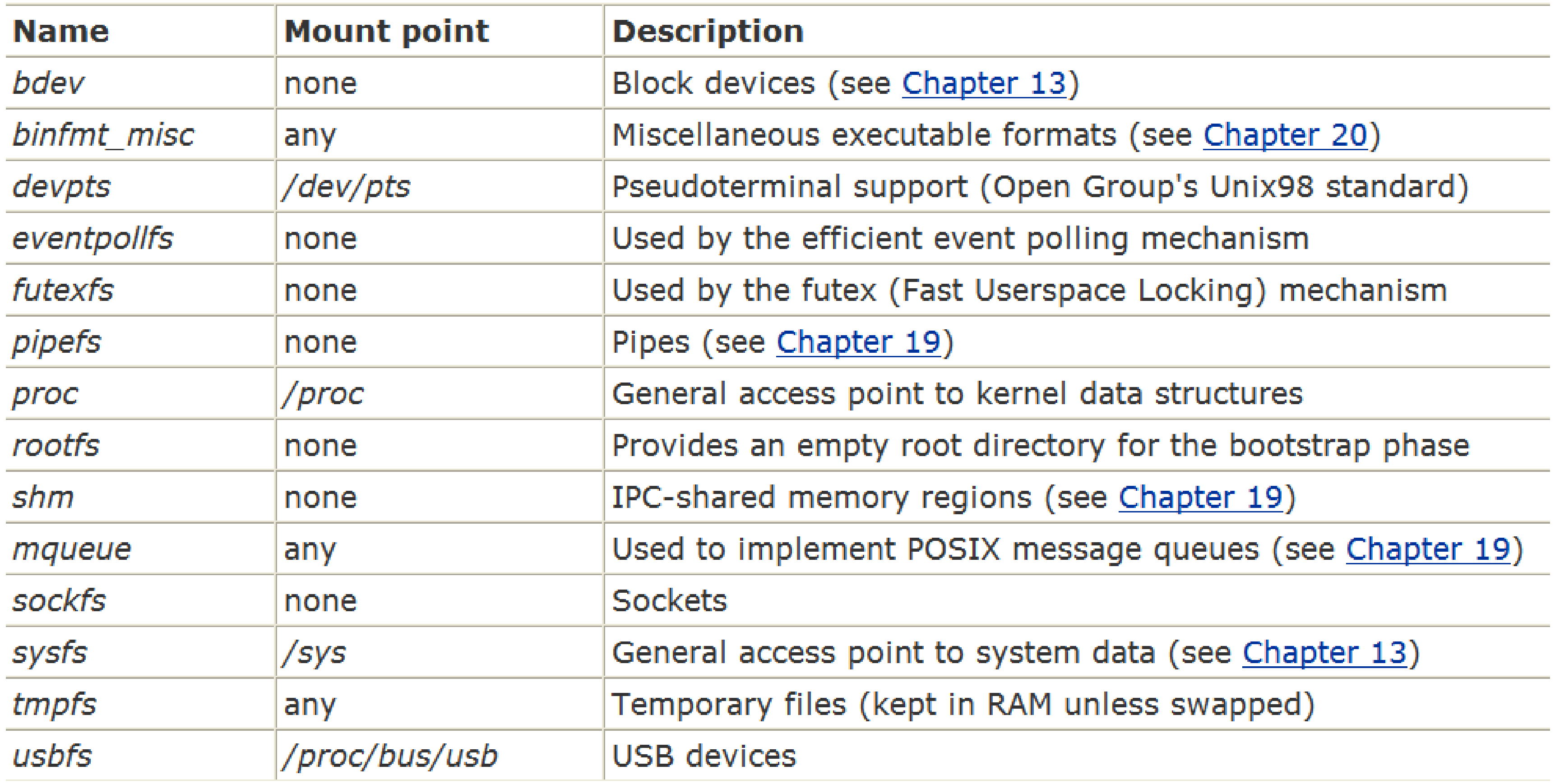
- a **generic block device layer** exists between the filesystem and the actual device driver---provides a layer of abstraction that allows the <u>implementation of the filesystem-specific code</u> to be independent of the specific device it eventually accesses.

Linux file system types

- Block device-based file systems
  - Build on top of a block device, manage memory space available in a local disk or in some other device that emulates a disk (such as a USB flash drive)
  - Ext3, ntfs…
- Network-based file systems
  - Build on top of network protocols, allow easy access to files included in filesystems belonging to other networked computers
  - Nfs, smb…
- Virtual/pseudo file systems
  - Build in memory, do not manage disk space, either locally or remotely
  - Proc, sysfs…
- Distributed file systems
  - GFS, HDFS…app layer

Linux file system types

special filesystems may provide an easy way for system programs and administrators to manipulate the data structures of the kernel and to implement special features of the operating system.

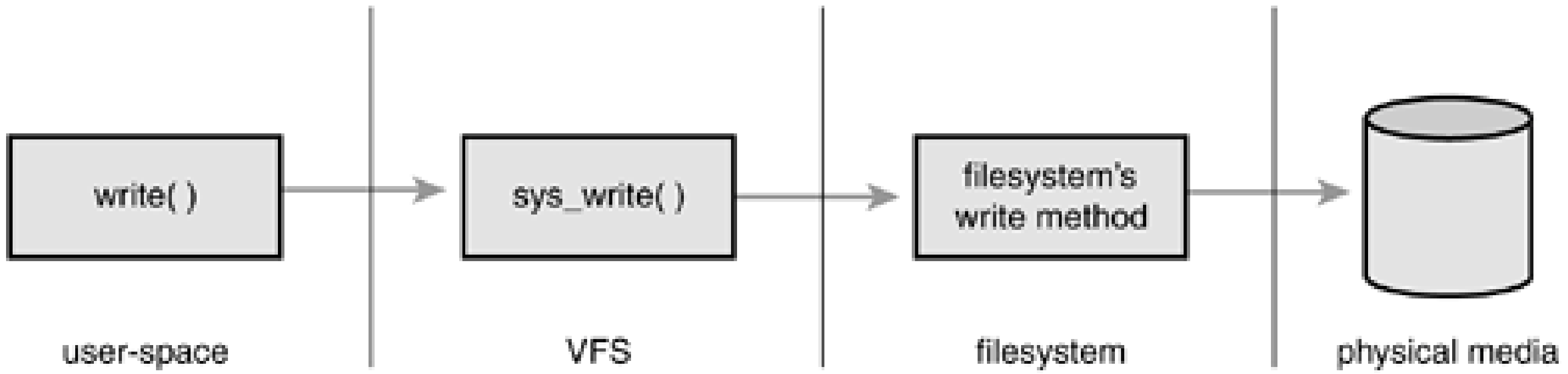


Name	Mount point	Description
bdev	none	Block devices (see Chapter 13)
binfmt_misc	any	Miscellaneous executable formats (see Chapter 20)
devpts	*/dev/pts*	Pseudoterminal support (Open Group's Unix98 standard)
eventpollfs	none	Used by the efficient event polling mechanism
futexfs	none	Used by the futex (Fast Userspace Locking) mechanism
pipefs	none	Pipes (see Chapter 19)
proc	*/proc*	General access point to kernel data structures
rootfs	none	Provides an empty root directory for the bootstrap phase
shm	none	IPC-shared memory regions (see Chapter 19)
mqueue	any	Used to implement POSIX message queues (see Chapter 19)
sockfs	none	Sockets
sysfs	*/sys*	General access point to system data (see Chapter 13)
tmpfs	any	Temporary files (kept in RAM unless swapped)
usbfs	*/proc/bus/usb*	USB devices

VFS layer

- Can be looked as Object-oriented
  - For reasons of efficiency, Linux is not wholy coded in an object-oriented language such as C++.
  - Objects are therefore implemented as plain C data structures with some fields pointing to functions that correspond to the object's methods
- Unified file system model
  - Traditional Unix like
  - FS implementation should be converted to the model if necessary
- Purpose
  - Unified user interface
  - Abstraction for all different file system implementations
- Responsibilities
  - System calls
  - Manage file systems data structures
  - Interact with different filesystem implementations

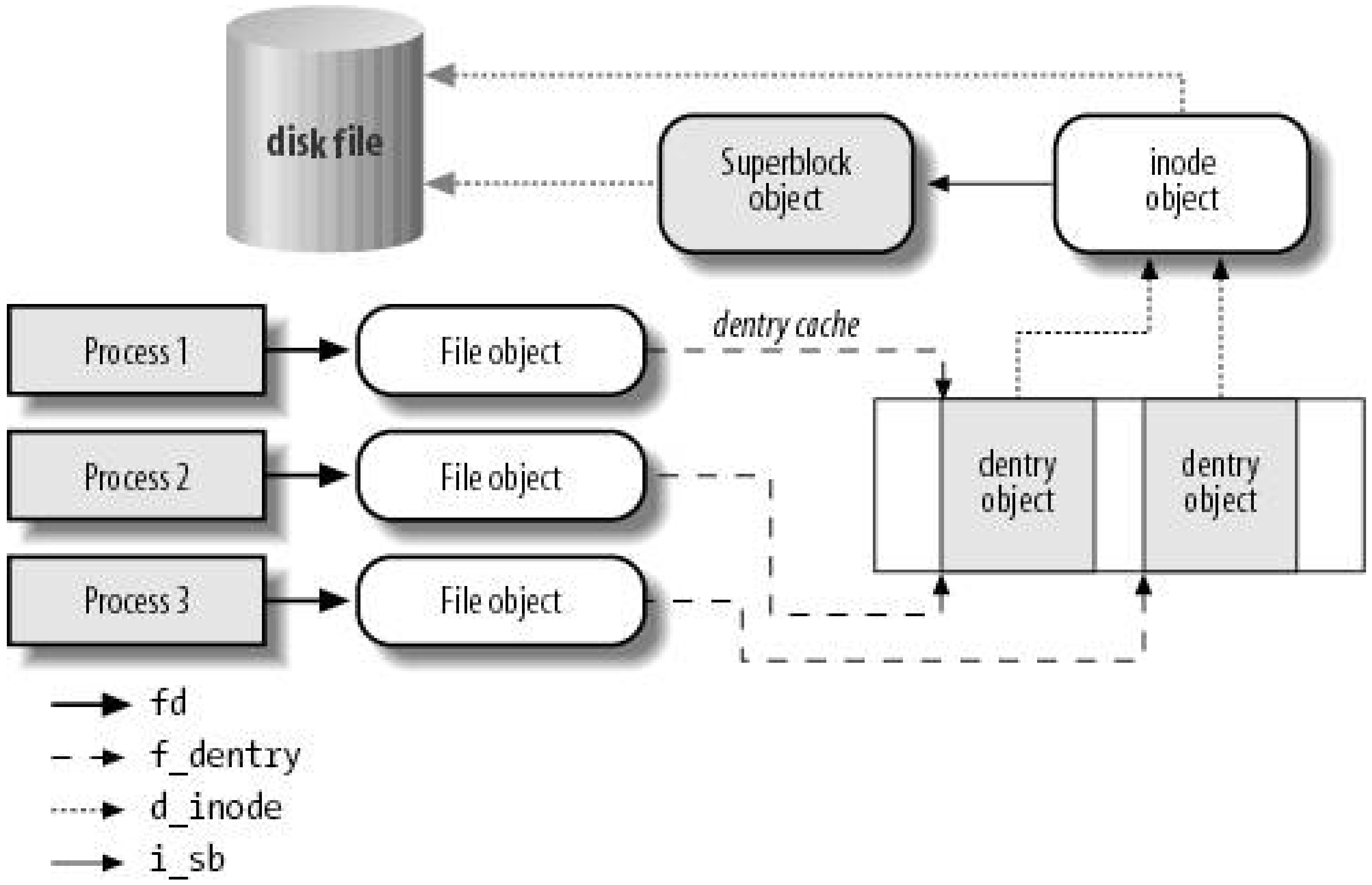
Filesystem Abstraction Layer



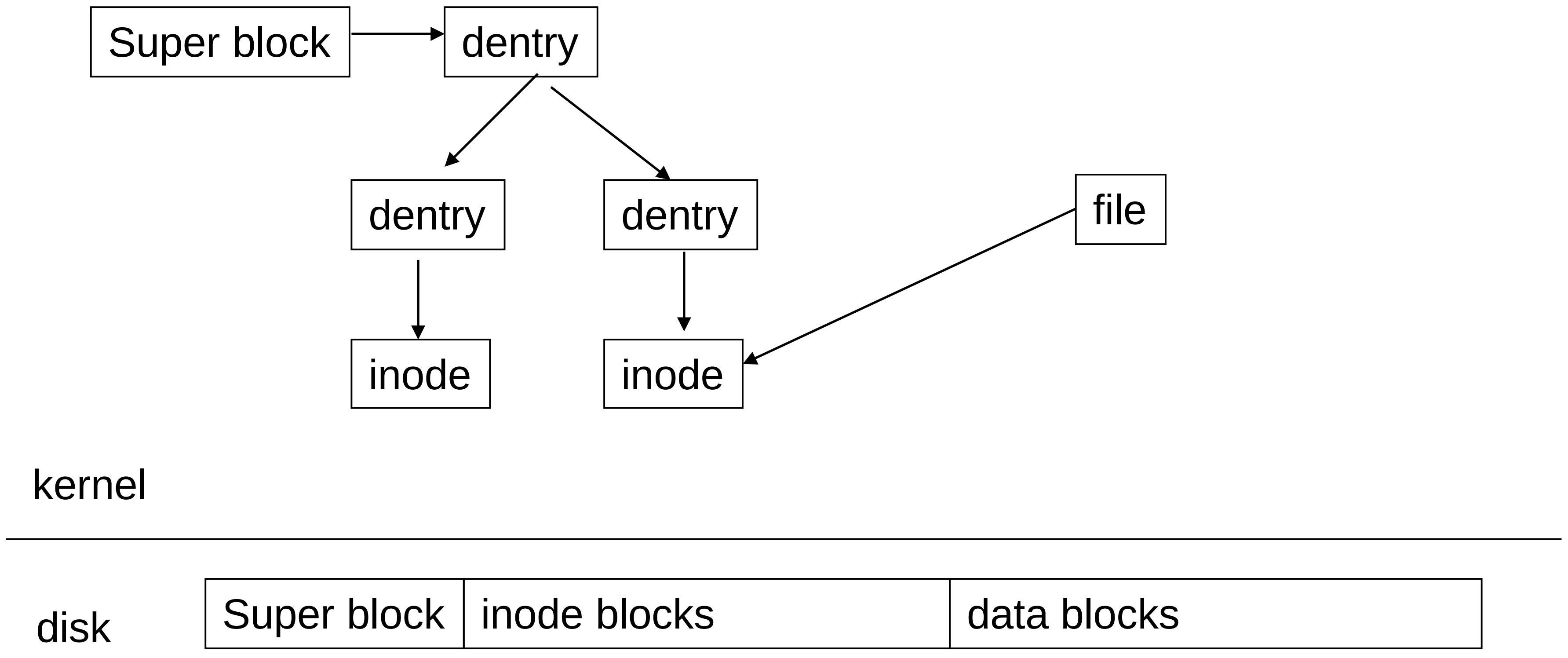
VFS Data Structures

- Important: hard disk VS. kernel
- Superblock
  - Info about a file system instance
  - May have disk counterpart (e.g. filesystem control block)
- Inode
  - Info about a file
  - May have disk counterpart (e.g. file control block)
- Dentry
  - about directory tree or about a name path
  - Each disk-based filesystem stores this information in its own particular way on disk
- File
  - info about the interaction between an open file and a process
  - exists only in <u>**kernel memory**</u> during the period when a process has the file open

虚拟文件系统中
各种对象之间的关系



Disk vs Kernel

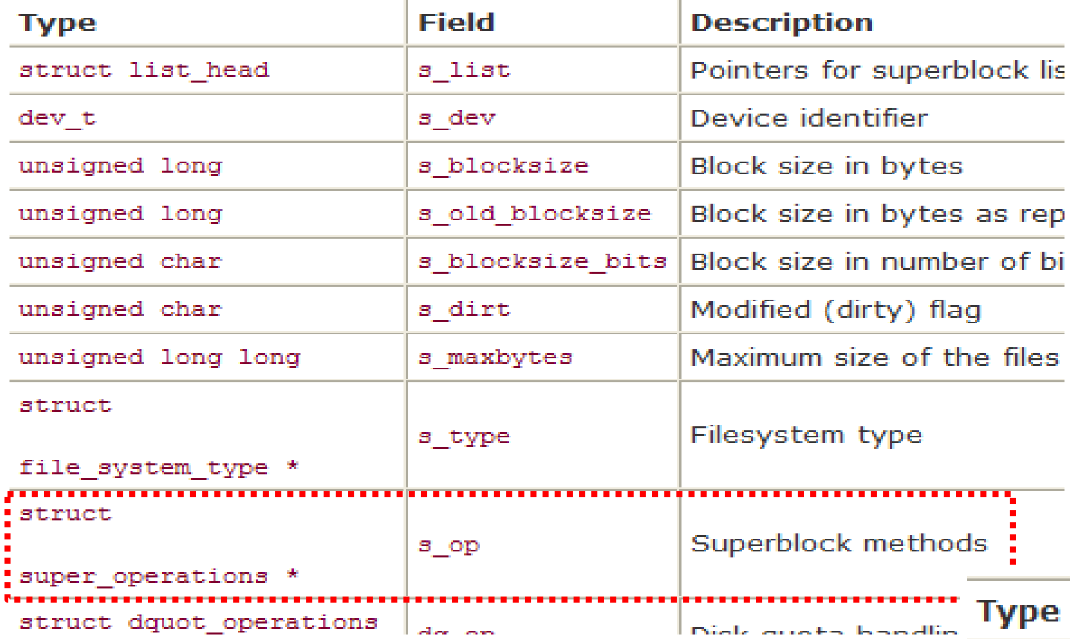


VFS objects common ground

- Reference counter for each object
  - Share (open an object several time)
  - State transition
- Cache
  - Cache objects in memory
  - For performance consideration
- Abstract objects
  - The object might not equal to the corresponding one of specific fs implementation (disk)
  - Specific fs implementation should convert their disk one to kernel one
- Object and its operations
  - For each object
  - Classical object-oriented

VFS object methods

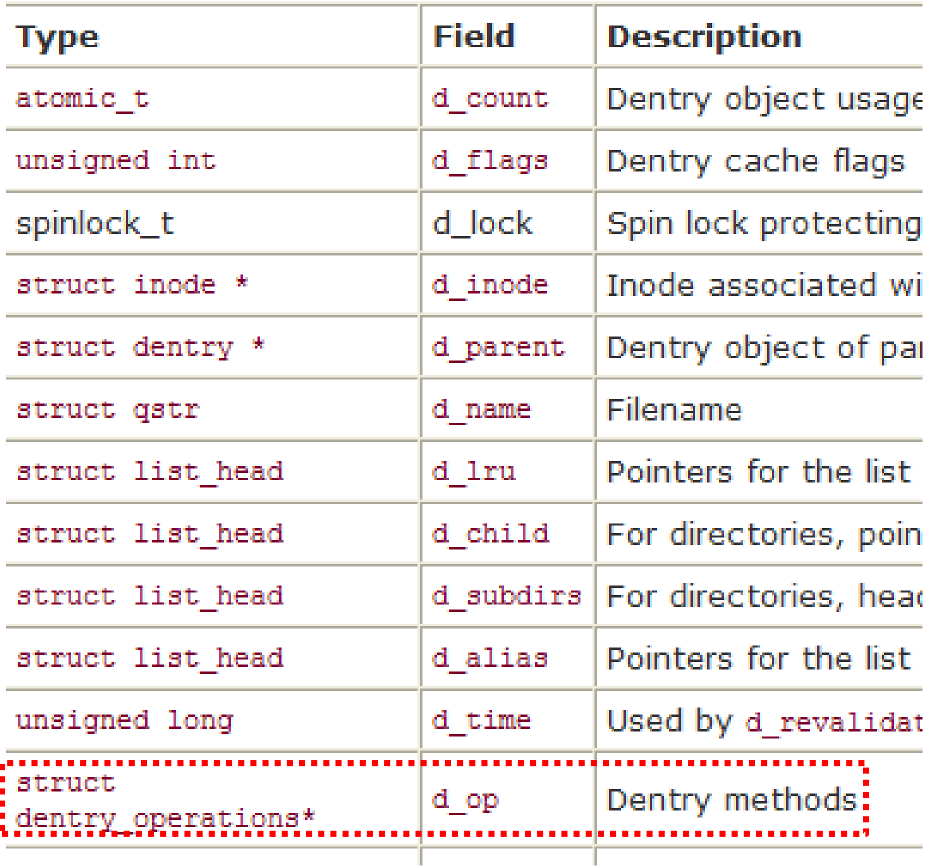
- An operation table for each object
  - Function pointers
  - File system specific or maybe object specific
- Link VFS with specific file system implementation
  - Implemented by specific file system
  - Called in VFS



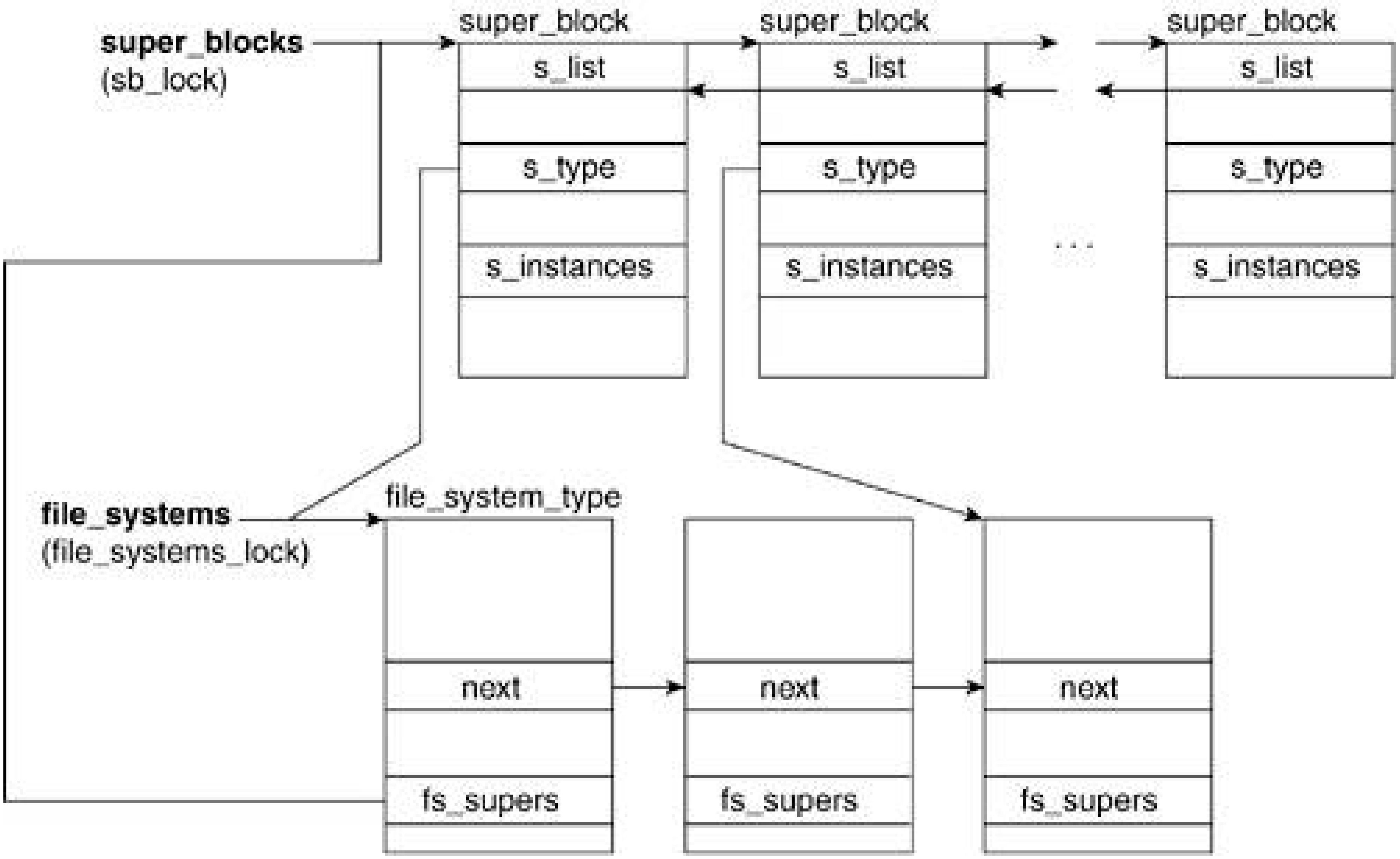
Type	Field	Description
struct list_head	s_list	Pointers for superblock lis
dev_t	s_dev	Device identifier
unsigned long	s_blocksize	Block size in bytes
unsigned long	s_old_blocksize	Block size in bytes as rep
unsigned char	s_blocksize_bits	Block size in number of bi
unsigned char	s_dirt	Modified (dirty) flag
unsigned long long	s_maxbytes	Maximum size of the files
struct file_system_type *	s_type	Filesystem type
struct super_operations *	s_op	Superblock methods
struct dquot_operations	dq_op	Disk quota handlin

- struct inode
- Main field
  - i_ino, i_sb(inode number & superblock)
  - i_mapping(address space)
  - i_fop(File operations)
  - i_op(Inode operations)
  - i_hash, i_sb_list (lists)
  - i_uid, i_gid, i_mode,i_atime(attributes)
  - i_count (reference count)

struct file
f_op (file operations)
f_dentry (file's dentry)
f_mode/f_owner (file attributes)
fu_list (link to superblock, remount should check opened files)
f_pos (read/write position)
f_count (reference count)
…



Type	Field	Description
atomic_t	d_count	Dentry object usage
unsigned int	d_flags	Dentry cache flags
spinlock_t	d_lock	Spin lock protecting
struct inode *	d_inode	Inode associated wi
struct dentry *	d_parent	Dentry object of pai
struct qstr	d_name	Filename
struct list_head	d_lru	Pointers for the list
struct list_head	d_child	For directories, poin
struct list_head	d_subdirs	For directories, heac
struct list_head	d_alias	Pointers for the list
unsigned long	d_time	Used by d_revalidat
struct dentry_operations*	d_op	Dentry methods



在系统中 superblock 结构是以链表的形式存放
File_system_type 是以链表存放

File system type

- About a file system implementation
  - struct file_system_type
  - Multiple fs instances per file_system_type
- Main fields
  - name (fs name, like ‘ext3’)
  - fs_flags (flags, like ‘FS_REQUIRES_DEV’)
  - mount (called when mounting a new fs instance)
  - kill_sb (called when unmounting a fs instance)
  - fs_supers(link to a list for all fs instances of the type)
  - owner(the module holding the fs implementation)

```
struct file_system_type {
                const char *name;
                int fs_flags;
#define FS_REQUIRES_DEV                 1
#define FS_BINARY_MOUNTDATA             2
#define FS_HAS_SUBTYPE                  4
#define FS_USERNS_MOUNT                                 8               /* Can be mounted by
                                                                          userns root */
#define FS_DISALLOW_NOTIFY_PERM                         16              /*Disable fanotify
                                                                         permission events */
#define FS_RENAME_DOES_D_MOVE                           32768 /*FS will handle
d_move()
                                                                  during rename() internally.
*/
                int (*init_fs_context)(struct fs_context *);
                const struct fs_parameter_spec *parameters;
                struct dentry *(*mount) (struct file_system_type *, int,
                                       const char *, void *);
                void (*kill_sb) (struct super_block *);
                struct module *owner;
                struct file_system_type * next;
                struct hlist_head fs_supers;

                ……
};
```

struct file_system_type

When a request is made to mount a device onto a directory in your filespace, the VFS will call the appropriate **mount()** method for the specific filesystem. **The dentry for the mount point will then be updated to point to the root inode for the new filesystem**.

mount: the method to call when a new instance of this filesystem should be mounted

struct file_system_type

- The mount field points to the filesystem-type-dependent function that **allocates a new superblock object and initializes it** (**if necessary, by reading a disk**).
- The **kill_sb** field points to the function that destroys a superblock.
- The fs_supers field represents the head (first dummy element) of a list of superblock objects corresponding to mounted filesystems of the given type
- fs_flags
  - FS_REQUIRES_DEV Every filesystem of this type must be located on a physical disk device.

Register a file system

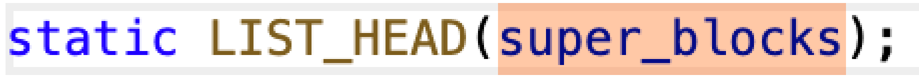
- File system implementation must register/unregister it into kernel
- int register_filesystem (struct **file_system_type** * fs);
  - inserts the corresponding file_system_type object into the file_system-type list (!=mount)
  - invoked
    - during system initialization (for every filesystem specified at compile time)
    - when a module implementing a filesystem is loaded (init_module)–can be unregistered (**unregister_filesystem()**)

static struct file_system_type *file_systems; /fs/filesystems.c

Superblock object

- **Information for a fs instance**
  - **Struct superblock**
  - **Maybe in disk. Kernel has a cached one for an instance**

fs/super.c



```
static LIST_HEAD(super_blocks);
```

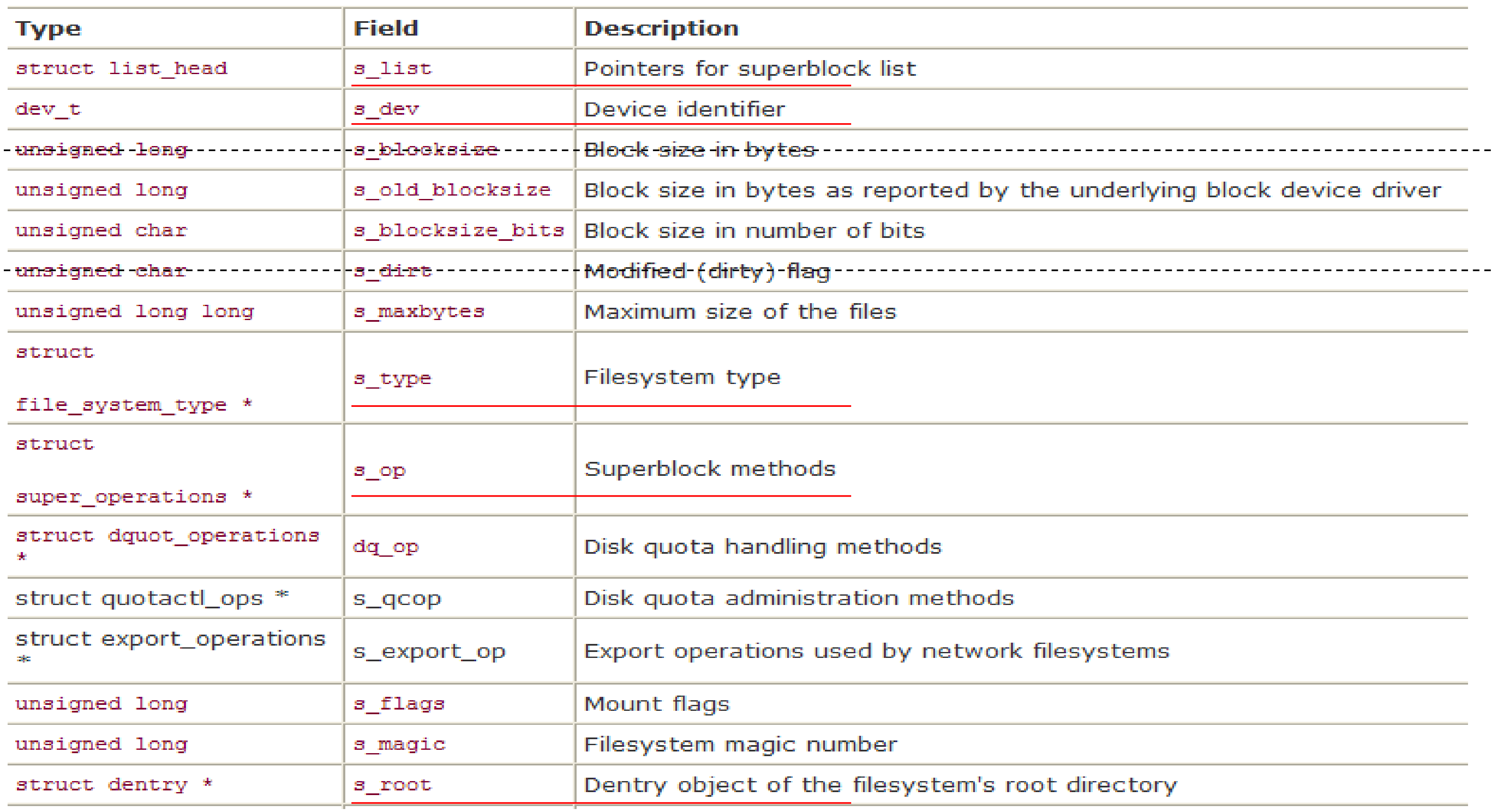
- **All superblock objects are linked in a circular doubly linked list**
  - **The first element of this list is represented by the super_blocks variable**
  - **the s_list field of the superblock object stores the pointers to the adjacent elements in the list**
  - **The sb_lock spin lock protects the list against concurrent accesses in multiprocessor systems.**
- **All superblock objects corresponding to a mounted filesystem of the given type are linked in a list**
  - **The fs_supers field (file_system_type) represents the head (first dummy element)**
  - **The backward and forward links of a list element are stored in the s_instances**

```
fs.h
struct super_block {
            struct list_head                    s_list;             指向超级块链表的指针
            dev_t                               s_dev;              设备标识符
            unsigned char                       s_blocksize_bits;   bit为单位的块大小
            unsigned long                       s_blocksize;        字节为单位的块大小
            loff_t                              s_maxbytes;         /* Max file size */文件的最大长度
            struct file_system_type             *s_type;            文件系统类型
            const struct super_operations               *s_op;      超级块方法
             ……
            struct hlist_node     s_instances;
            ……
            struct dentry                       *s_root;            文件系统根目录的目录项对象
            struct rw_semaphore                 s_umount;
            …
}


super.c
static DEFINE_SPINLOCK(sb_lock);
```

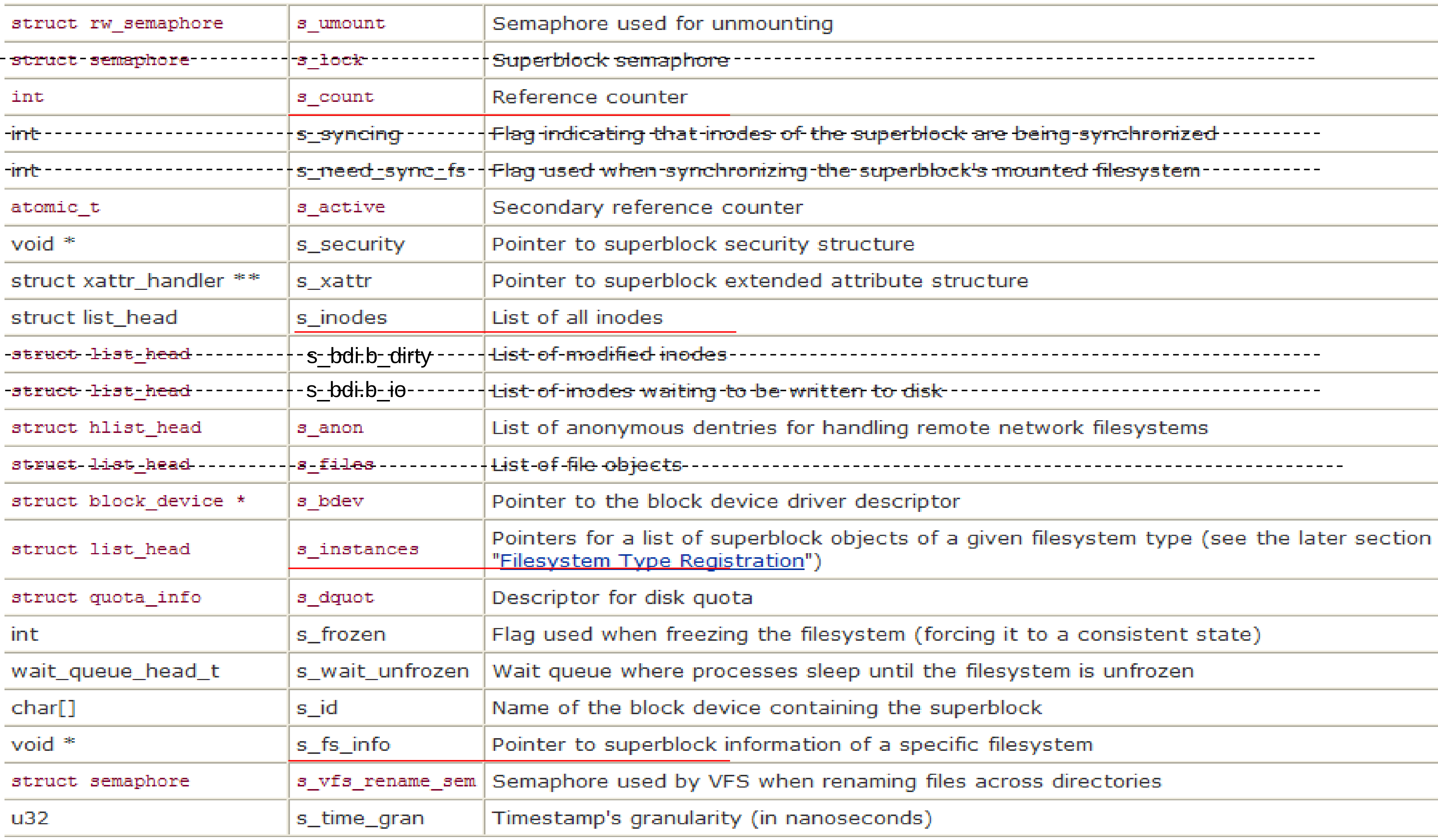
The fields of the superblock object

<linux/fs.h>



Type	Field	Description
struct list_head	s_list	Pointers for superblock list
dev_t	s_dev	Device identifier
~~unsigned long~~	~~s_blocksize~~	~~Block size in bytes~~
unsigned long	s_old_blocksize	Block size in bytes as reported by the underlying block device driver
unsigned char	s_blocksize_bits	Block size in number of bits
~~unsigned char~~	~~s_dirt~~	~~Modified (dirty) flag~~
unsigned long long	s_maxbytes	Maximum size of the files
struct file_system_type *	s_type	Filesystem type
struct super_operations *	s_op	Superblock methods
struct dquot_operations *	dq_op	Disk quota handling methods
struct quotactl_ops *	s_qcop	Disk quota administration methods
struct export_operations *	s_export_op	Export operations used by network filesystems
unsigned long	s_flags	Mount flags
unsigned long	s_magic	Filesystem magic number
struct dentry *	s_root	Dentry object of the filesystem's root directory
const struct dentry_operations *	**s_d_op;**	**default d_op for dentries**

The fields of the superblock object



struct rw_semaphore	s_umount	Semaphore used for unmounting
~~struct semaphore~~	~~s_lock~~	~~Superblock semaphore~~
int	s_count	Reference counter
~~int~~	~~s_syncing~~	~~Flag indicating that inodes of the superblock are being synchronized~~
~~int~~	~~s_need_sync_fs~~	~~Flag used when synchronizing the superblock's mounted filesystem~~
atomic_t	s_active	Secondary reference counter
void *	s_security	Pointer to superblock security structure
struct xattr_handler **	s_xattr	Pointer to superblock extended attribute structure
struct list_head	s_inodes	List of all inodes
~~struct list_head~~	s_bdi.b_dirty	~~List of modified inodes~~
~~struct list_head~~	s_bdi.b_io	~~List of inodes waiting to be written to disk~~
struct hlist_head	s_anon	List of anonymous dentries for handling remote network filesystems
~~struct list_head~~	~~s_files~~	~~List of file objects~~
struct block_device *	s_bdev	Pointer to the block device driver descriptor
struct list_head	s_instances	Pointers for a list of superblock objects of a given filesystem type (see the later section "Filesystem Type Registration")
struct quota_info	s_dquot	Descriptor for disk quota
int	s_frozen	Flag used when freezing the filesystem (forcing it to a consistent state)
wait_queue_head_t	s_wait_unfrozen	Wait queue where processes sleep until the filesystem is unfrozen
char[]	s_id	Name of the block device containing the superblock
void *	s_fs_info	Pointer to superblock information of a specific filesystem
struct semaphore	s_vfs_rename_sem	Semaphore used by VFS when renaming files across directories
u32	s_time_gran	Timestamp's granularity (in nanoseconds)

Superblock operations

- Each specific filesystem can define its own superblock operations
- struct super_operations
  - write_super(read super is in filesystem type)
  - write_inode (write an inode to disk)
  - drop_inode (remove every reference to the inode and, if the inode no longer appears in any directory, invoke delete_inode)
  - …
- struct super_block’s **s_op** field

struct super_operations

```
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);

        void (*dirty_inode) (struct inode *, int flags);
        int (*write_inode) (struct inode *, struct writeback_control *wbc);
        int (*drop_inode) (struct inode *);
        void (*evict_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        int (*sync_fs)(struct super_block *sb, int wait);
        int (*freeze_super) (struct super_block *);
        int (*freeze_fs) (struct super_block *);
        int (*thaw_super) (struct super_block *);
        int (*unfreeze_fs) (struct super_block *);
        int (*statfs) (struct dentry *, struct kstatfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*umount_begin) (struct super_block *);
        ……
```

<linux/fs.h>

Superblock object

- The code for **creating**, **managing**, and **destroying** superblock objects lives in fs/super.c.
- A superblock object is created and initialized via the sget function. /fs/super.c
  - When mounted, a filesystem invokes this function, reads its superblock off of the disk, and fills in its superblock object.

Inode object

- Inode represents a file, store file’s info
- Disk inode resides on disk, and kernel has a cached one
- Inode number is unique for a fs instance
- Inode cache：Recently accessed unused inode will be cached for performance purpose, organized by hashtable, based on slab allocator

```
void __init inode_init(void)
{
            /* inode slab cache */
            inode_cachep = kmem_cache_create("inode_cache",
                                                 sizeof(struct inode),
                                                 0,

(SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|

SLAB_MEM_SPREAD|SLAB_ACCOUNT),
                                                 init_once);

            /* Hash may have been set up in inode_init_early */
            if (!hashdist)
                        return;

            inode_hashtable =
                        alloc_large_system_hash("Inode-cache",
                                                sizeof(struct hlist_head),
                                                ihash_entries,    14,
                                                HASH_ZERO,       &i_hash_shift,
                                                &i_hash_mask,
                                                0,   0);
}
```

/fs/ext2/ialloc.c

```
struct inode *ext2_new_inode(struct inode *dir,
                umode_t mode, const struct qstr *qstr)
{
        ……
        inode = new_inode(sb);
        ……
}
```

$$\begin{tikzcd}
\texttt{inode = new\_inode(sb);} \arrow[d] \\
\texttt{new\_inode\_pseudo(sb)} \arrow[d] \\
\texttt{alloc\_inode(sb)} \arrow[d] \\
\texttt{inode = kmem\_cache\_alloc(inode\_cachep, GFP\_KERNEL);}
\end{tikzcd}$$

Inode structure

- struct inode
- Main field
  - i_ino, i_sb(inode number & superblock)
  - i_mapping(address space)
  - i_fop(File operations)
  - i_op(Inode operations)
  - i_hash, i_sb_list (lists)
  - i_uid, i_gid, i_mode,i_atime(attributes)
  - i_count (reference count)

ext2_mount() → ext2_fill_super() → ext2_iget() → ext2_set_file_ops()

```
void ext2_set_file_ops(struct inode *inode)
{
        inode->i_op = &ext2_file_inode_operations;
        inode->i_fop = &ext2_file_operations;
        if (IS_DAX(inode))
                inode->i_mapping->a_ops = &ext2_dax_aops;
        else if (test_opt(inode->i_sb, NOBH))
                inode->i_mapping->a_ops = &ext2_nobh_aops;
        else
                inode->i_mapping->a_ops = &ext2_aops;
}
```

Inode lists

- Superblock’s inodes
  - each inode object is included in a per-filesystem doubly linked circular list
    - headed at the **s_inodes** field of the **superblock** object
    - i_sb_list field of the inode object stores the pointers for the adjacent elements in this list
- Inode hashtable
  - Fast search a cached inode
  - Global **inode_hashtable**
  - Hashed by i_ino & i_sb

Inode lists

- Free inodes
  - headed at the **s_inode_lru** field of the **superblock** object
  - **i_lru** field of the inode object stores the pointers for the adjacent elements in this list

- Dirty inodes
  - Inodes waiting for writeback
  - **s_inodes_wb** of struct **super_block**
  - **i_wb_list** field of the inode object stores the pointers for the adjacent elements in this list

Inode operations

- struct inode_operations
  - lookup (lookup an inode in a parent dir)
  - create (create a new file)
  - mkdir/rmdir(create/delete dir)
  - Link/unlink/symlink (create hard link/symbol link)
  - …
- inode kernel APIs
  - iget(): get an inode (find in cache, if no then create from disk)
  - ilookup(): search for an inode in the inode cache
  - …

ext2_iget() →

```
/**
 * iget_locked - obtain an inode from a mounted file system
 * @sb:			super block of file system
 * @ino:	inode number to get
 *
 * Search for the inode specified by @ino in the inode cache and if present
 * return it with an increased reference count. This is for file systems
 * where the inode number is sufficient for unique identification of an inode.
 *
 * If the inode is not in cache, allocate a new inode and return it locked,
 * hashed, and with the I_NEW flag set.  The file system gets to fill it in
 * before unlocking it via unlock_new_inode().
 */
struct inode *iget_locked(struct super_block *sb, unsigned long ino)
{
		struct hlist_head *head = inode_hashtable + hash(sb, ino);
		struct inode *inode;


		…
}
```

```
/*** ilookup5_nowait - search for an inode in the inode cache
 * @sb:              super block of file system to search
 * @hashval:         hash value (usually inode number) to search for
 * @test:  callback used for comparisons between inodes
 * @data:opaque data pointer to pass to @test
 * Search for the inode specified by @hashval and @data in the inode cache.
 * If the inode is in the cache, the inode is returned with an incremented
 * reference count.
 * Note: I_NEW is not waited upon so you have to be very careful what you do
 * with the returned inode.  You probably should be using ilookup5() instead.
 * Note2: @test is called with the inode_hash_lock held, so can't sleep.
 */
struct inode *ilookup5_nowait(struct super_block *sb, unsigned long hashval,
                  int (*test)(struct inode *, void *), void *data)
{
        struct hlist_head *head = inode_hashtable + hash(sb, hashval);
        struct inode *inode;

        spin_lock(&inode_hash_lock);
        inode = find_inode(sb, head, test, data);
        spin_unlock(&inode_hash_lock);

        return inode;
}
```

Dentry objects

- VFS **considers each directory a file**
- Once **a directory entry** is read into memory, however, it is transformed by the VFS into a **dentry** object based on the dentry structure.
- The kernel creates a dentry object for every component of a pathname that a process looks up; the dentry object associates the component to its corresponding inode.
  - For example, when looking up the */tmp/test* pathname, the kernel creates a dentry object for the / root directory, a second dentry object for the *tmp* entry of the root directory, and a third dentry object for the *test* entry of the */tmp* directory.
  - 使用 kmem_cache_alloc( ) (Allocating a Slab Object) 和 kmem_cache_free( ) 创建和释放 dentry objects

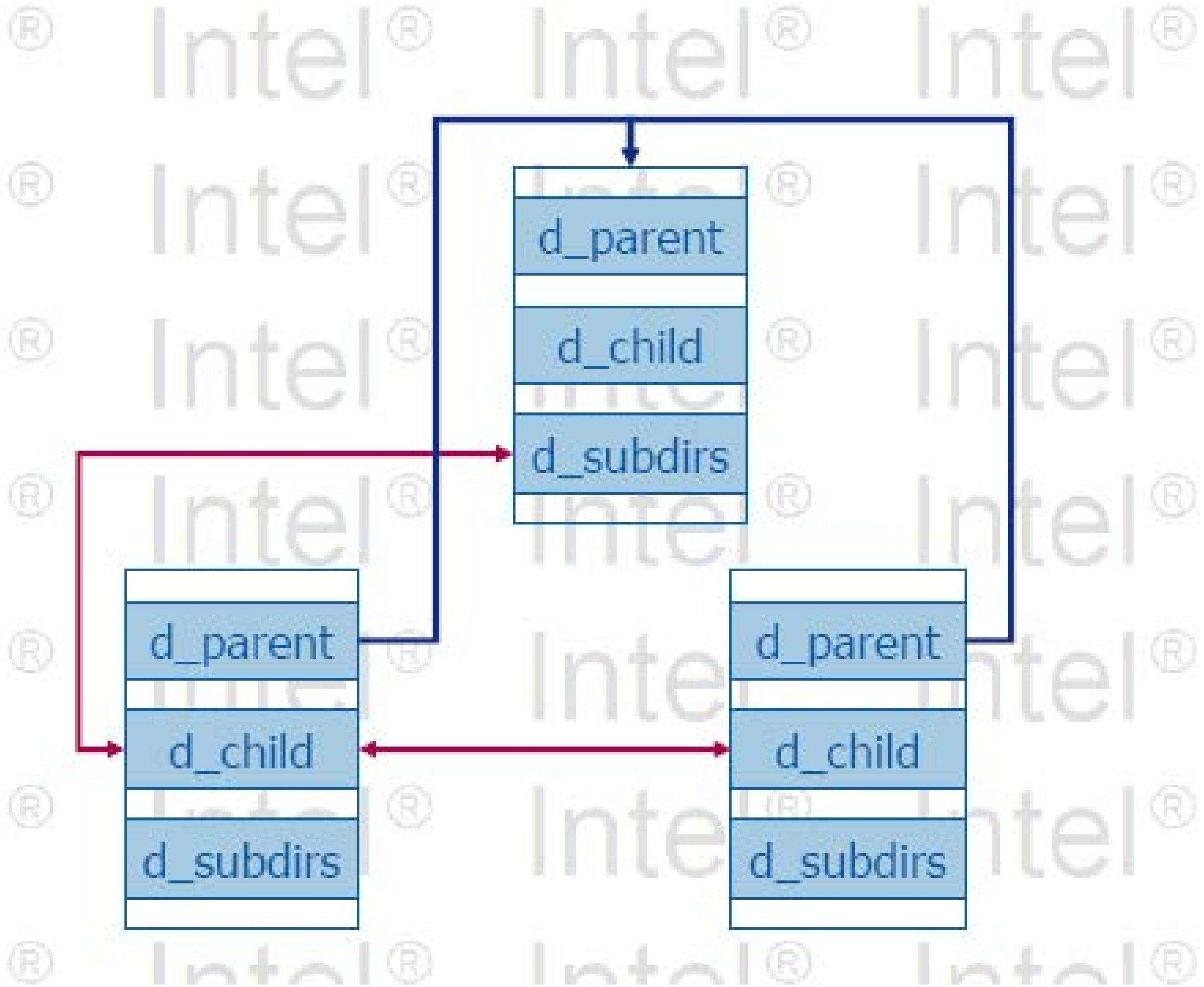
```
struct dentry {
        /* RCU lookup touched fields */
        unsigned int d_flags;                          /* protected by d_lock */
        seqcount_t d_seq;                /* per dentry seqlock */
        struct hlist_bl_node d_hash; /* lookup hash list */
        struct dentry *d_parent;         /* parent directory */
        struct qstr d_name;
        struct inode *d_inode;           /* Where the name belongs to - NULL is negative */
        unsigned char d_iname[DNAME_INLINE_LEN];    /* small names */
        /* Ref lookup also touches following */
        struct lockref d_lockref;        /* per-dentry lock and refcount */
        const struct dentry_operations *d_op;
        struct super_block *d_sb;        /* The root of the dentry tree */
        unsigned long d_time;                          /* used by d_revalidate */
        void *d_fsdata;                                /* fs-specific data */
        struct list_head d_lru;                        /* LRU list */
        struct list_head d_child;        /* child of parent list */
        struct list_head d_subdirs;      /* our children */
        /* * d_alias and d_rcu can share memory */
        union {
                struct hlist_node d_alias;     /* inode alias list */
                struct rcu_head d_rcu;
        } d_u;
};
```

<linux/dcache.h>

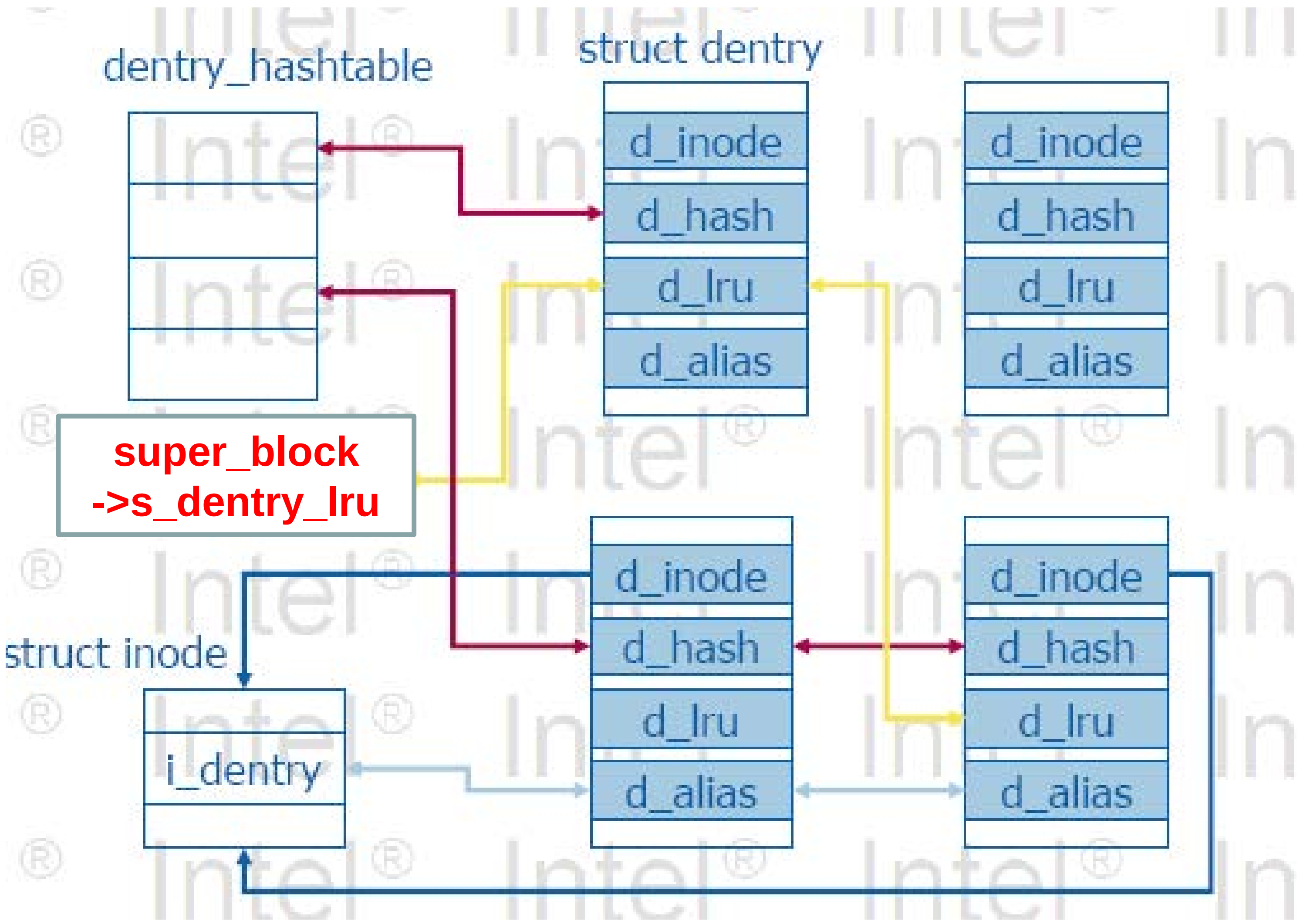
Dentry list

- Tree layout(d_parent, d_subdirs, d_child)
- Inuse list(d_alias) : Pointers for the list of dentries associated with the same inode (alias)
- Unused and negative dentry list(d_lru)
- Hashtable(hash collision list, d_hash)

Dentry tree layout



Dentry lists



Dentry operations

- Dentry_operations
  - d_hash (return hash value of a file name)
  - d_compare (wether file name and dentry match)
  - d_delete(delete a dentry, when d_count = 0)
  - …
- Dentry kernel APIs
  - d_add(): establish the relationships between inode and dentry
  - d_alloc(): allocate and initialize a dentry struct
  - d_lookup(dentry, qstr): search a dentry named qstr in hash list
  - ……

The dentry Cache

- Because reading a directory entry from disk and constructing the corresponding dentry object requires considerable time, it makes sense to keep in memory dentry objects that you've finished with but might need later
- consist of two kinds of data structures:
  - A set of dentry objects in the in-use, unused, or negative state
    - Lists of "used" dentries
    - LRU list of unused and negative dentry objects
  - A hash table to derive the dentry object associated with a given filename and a given directory quickly.
- The dentry cache also acts as a controller for an inode cache. As long as the dentry is cached, the corresponding inodes are cached, too.

Dentry objects' states

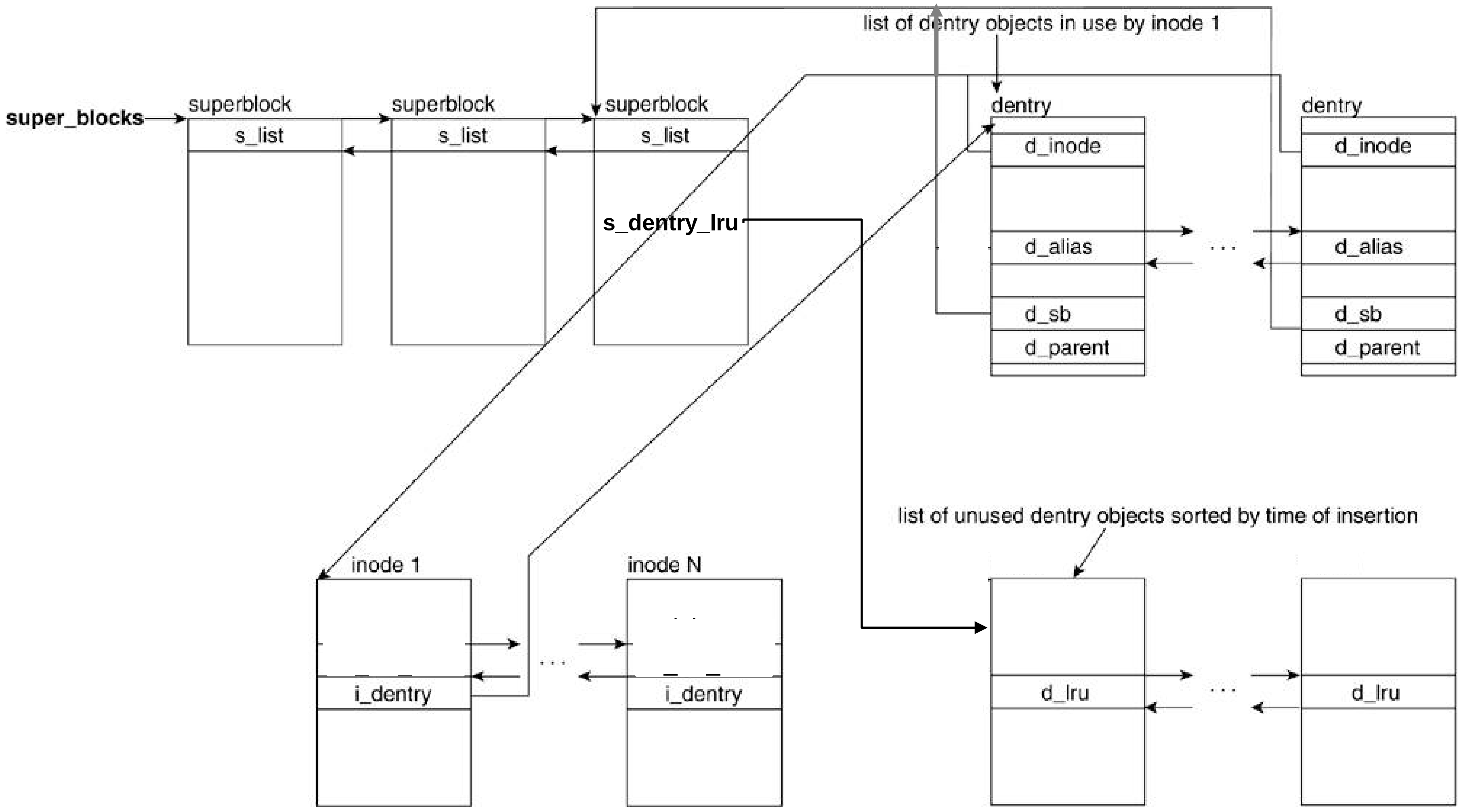
- Each dentry object may be in one of four states:
  - Free
    - The dentry object contains no valid information and is not used by the VFS
    - The corresponding memory area is handled by the slab allocator.
  - Unused
    - The dentry object is not currently used by the kernel, but contains **valid information.**
    - d_count = 0
    - d_inode points to the associated inode
    - included in a doubly linked "Least Recently Used" list sorted by time of insertion.
      - When the dentry cache has to shrink, the kernel removes elements from the list
      - The addresses of the first and last elements of the LRU list are stored in the next and prev fields of the **super_block->s_dentry_lru** variable of type list_head
      - The **d_lru** field of the dentry object contains pointers to the adjacent dentries in the list.

Dentry objects' states

- In use
  - The dentry object is currently used by the kernel, contains valid information and cannot be discarded.
  - d_count > 0
  - d_inode points to the associated inode object
  - doubly linked list
    - specified by the **i_dentry** field of the corresponding inode object
    - adjacent elements: **d_alias** field of the dentry object
    - In tree layout
- Negative
  - **The inode associated with the dentry does not exist,** d_inode is set to NULL
    - either because the corresponding disk inode has been deleted
    - or because the dentry object was created by resolving a pathname of a nonexistent file
  - the dentry object still remains in the dentry cache, so that further lookup operations to the same file pathname can be quickly resolved

An "in use" dentry object may become "negative" when the last hard link to the corresponding file is deleted. In this case, the dentry object is moved into the LRU list of unused dentries.

Superblock 和 i-node以及 dentry 的关系



File object

- Describes how a process interacts with an opened file
- Resides in kernel
- struct file
  - f_op (file operations)
  - f_dentry (file’s dentry)
  - f_mode/f_owner (file attributes)
  - fu_list (link to superblock, remount should check opened files)
  - f_pos (read/write position)
  - f_count (reference count)
  - …

File operations

- Struct file_operations
  - open
  - write/read
  - llseek (set file read/write position)
  - mmap (called when doing mmap syscall, memory maps the given file onto the given address space)
  - release (called when closing a file)
  - poll (called when doing select/poll syscall)
  - Ioctl
  - …

```
struct file_operations {
        struct module *owner;           loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*mremap)(struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);     int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);  int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);          int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,  loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
#endif };
```

Structures associated with a process

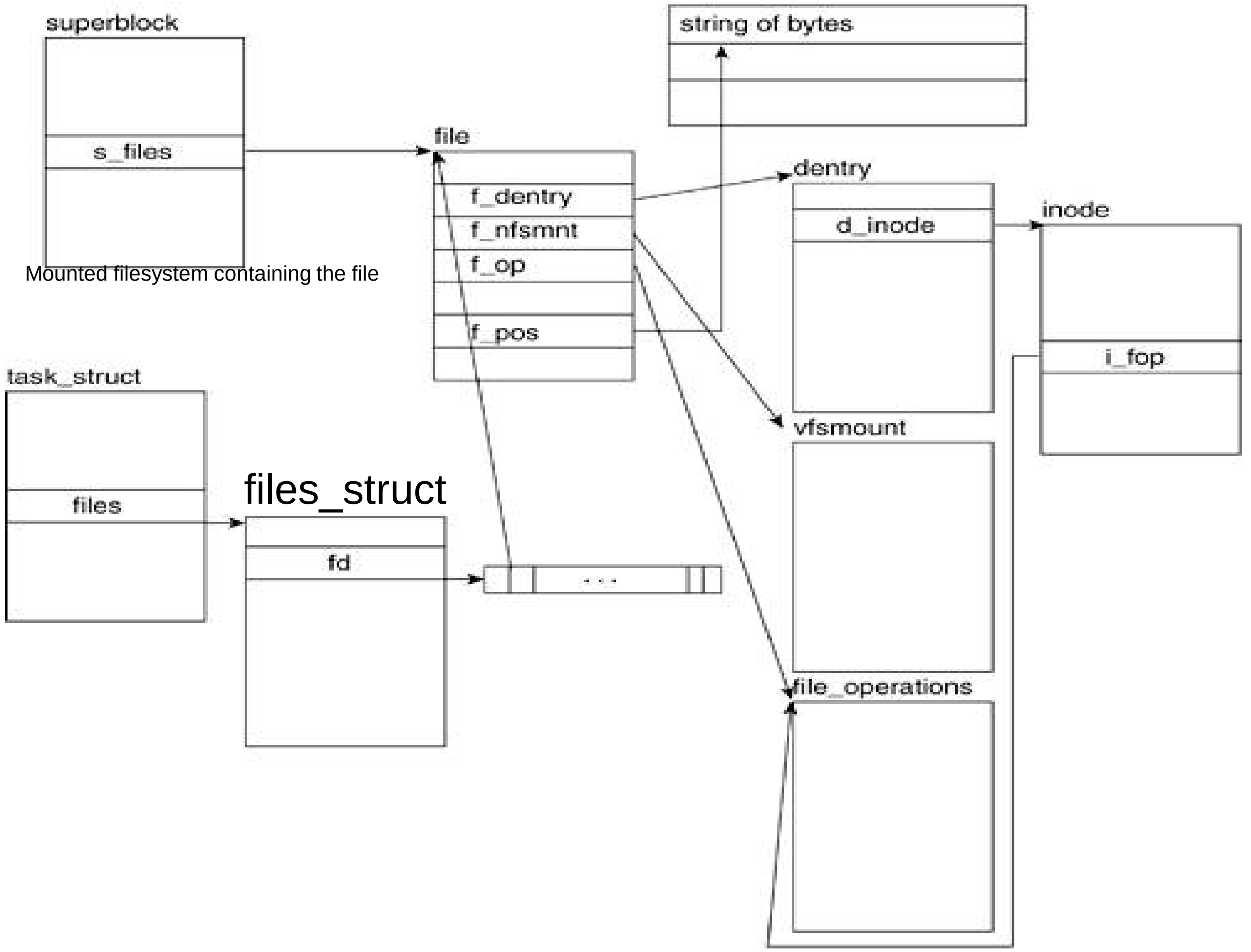
- struct task_struct has two fields related with fs
  - struct fs_struct *fs
  - struct files_struct *files
- struct files_struct
  - About opened files of a task
- Struct fs_struct
  - About task's filesystem environment
  - Root mountpoint (vfsmount), root dentry: Mounted filesystem object of the **root directory**
  - Pwd mountpoint (vfsmount), pwd dentry: Mounted filesystem object of the **current working directory**

files_struct

```
struct fdtable {
        unsigned int max_fds;
        struct file __rcu ** fd;      /* current fd array */
        fd_set *close_on_exec;
        fd_set *open_fds;
        struct rcu_head rcu;
        struct fdtable *next;
};
```

```
struct files_struct {
  /*
   * read mostly part
   */
        atomic_t count;
        struct fdtable __rcu *fdt;
        struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
        spinlock_t file_lock ____cacheline_aligned_in_smp;
        int next_fd;
        struct embedded_fd_set close_on_exec_init[1];
        struct embedded_fd_set open_fds_init[1];
        struct file __rcu * fd_array[NR_OPEN_DEFAULT];
};
```

File Objects



- A file object describes how a process interacts with a file it has opened
- The object is created when the file is opened and consists of a file structure
- Notice that file objects have **no corresponding image on disk**, and hence **no "dirty" field** is included in the file structure to specify that the file object has been modified.

vfsmount

- 每个装载的文件系统都对应一个vfsmount结构的实例。

```
50 struct vfsmount {
51         struct dentry *mnt_root;       /* root of the mounted tree */
52         struct super_block *mnt_sb;     /* pointer to superblock */
53         int mnt_flags;
54 };
```

mnt_root 指向其挂载树的根目录，mnt_sb指向对应的super_block
mnt_flags用于设置各种独立于文件系统的标志，nount.h中定义了其各种可能的取值

已经被mount的文件系统实例，需要查找vfsmount的cache，找到其挂载点信息，进行根目录重定向

```
struct mount {
        struct hlist_node mnt_hash;   //global hash mounted fs table
        struct mount *mnt_parent;  //parent mount struct
        struct dentry *mnt_mountpoint;  //mountpoint dentry of  parent fs
        struct vfsmount mnt;
        union {
                struct rcu_head mnt_rcu;
                struct llist_node mnt_llist;
        };
#ifdef CONFIG_SMP
        struct mnt_pcp __percpu *mnt_pcp;
#else
        int mnt_count;
        int mnt_writers;
#endif
        struct list_head mnt_mounts;      /* list of children, anchored here */
        struct list_head mnt_child;       /* and going through their mnt_child */
        ……
} __randomize_layout;
```

Pathname lookup

- kern_path(): Find the inode/dentry from a given pathname
  - Frequently used
- Start point dentry
  - Pathname starts ‘/’: fs->root
  - Else: fs->pwd
- Special handling
  - Symbolic links (check loops)
  - Mount points
  - Access permission
- The result of the pathname lookup
  - local variable nd of type nameidata

The fields of the nameidata data structure

```
struct nameidata {
        struct path     path;
        struct qstr     last;
        struct path     root;

        struct inode    *inode; /* path.dentry.d_inode */
        unsigned int    flags;
        unsigned        seq;
        int             last_type;
        unsigned        depth;
        char *saved_names[MAX_NESTED_LINKS + 1];

        /* Intent data */
        union {
                struct open_intent open;
        } intent;
};
```

```
struct path {
        struct vfsmount *mnt;
        struct dentry *dentry;
};
```

The dentry and mnt fields "describe" the file identified by the given pathname

- both objects should not be freed until the caller of path_lookup( ) finishes using them. Therefore, path_lookup( ) increases the usage counters of both objects. If the caller wants to release these objects, it invokes the path_release( ) function passing as parameter the address of a nameidata structure.

path_lookupat

- link_path_walk( )
  - the core of the pathname lookup operation
  - Parameters: name, nd
- Parse one name one loop
  - Check access permission (may_lookup)
  - Calculate name hash (used when look in the dentry cache hash table)
  - Check special file (eg. “.”)
  - Use walk_component check dentry cache or real disk
  - Check symbolic link

link_path_walk() → walk_component()→ lookup_fast→d_lookup

→ lookup_slow→ inode->i_op->lookup

ext2_fill_super() → ext2_iget() → ext2_set_file_ops()

```
void ext2_set_file_ops(struct inode *inode)
{
        inode->i_op = &ext2_file_inode_operations;
        inode->i_fop = &ext2_file_operations;
        if (IS_DAX(inode))
                inode->i_mapping->a_ops = &ext2_dax_aops;
        else if (test_opt(inode->i_sb, NOBH))
                inode->i_mapping->a_ops = &ext2_nobh_aops;
        else
                inode->i_mapping->a_ops = &ext2_aops;
}
```

Lookup example

```
struct dentry *ext3_lookup(struct inode * dir, struct
dentry *dentry, struct nameidata *nd)
{
… /* get inode number */
inode = ext3_iget(dir->i_sb, ino); /*Get inode of inode
    number*/
…
return d_splice_alias(inode, dentry); /* link dentry with inode */
}
```

Lookup example

- ext3_iget (Get inode of inode number)
  - Search **inode cache**
  - Or
    - allocate a new inode (sb->s_op->alloc_inode(), file system specific)
    - Read inode in (sb->s_op->read_inode(), file system specific)
- d_splice_alias (link dentry with inode)
  - Add dentry into hash list (so it can be found by cached lookup)

Path walk special handling

- Mount point – follow_mount()
  - checks whether is a mount point for some filesystem
    - nd->dentry->d_mounted
  - It's possible to mount multiple fs in the same directory
    - Fs mount is stacked
    - jump to the latest mounted file system on the dentry
- Symbolic link
  - Specific fs must support symbolic link
  - do_follow_link (check **loop**, **jump** to linked file. Uses inode)
  - operation's
    - follow_link() (called by the VFS to translate a symbolic link to the **inode** to which it points)
    - put_link() (Releases all temporary data structures allocated by the follow_link method to translate a symbolic link)

页缓存

- **Linux**采用基数树（**radix tree**）来管理页缓存中包含的页
- 内核提供了几个同步方案用于写回修改的数据

基数树

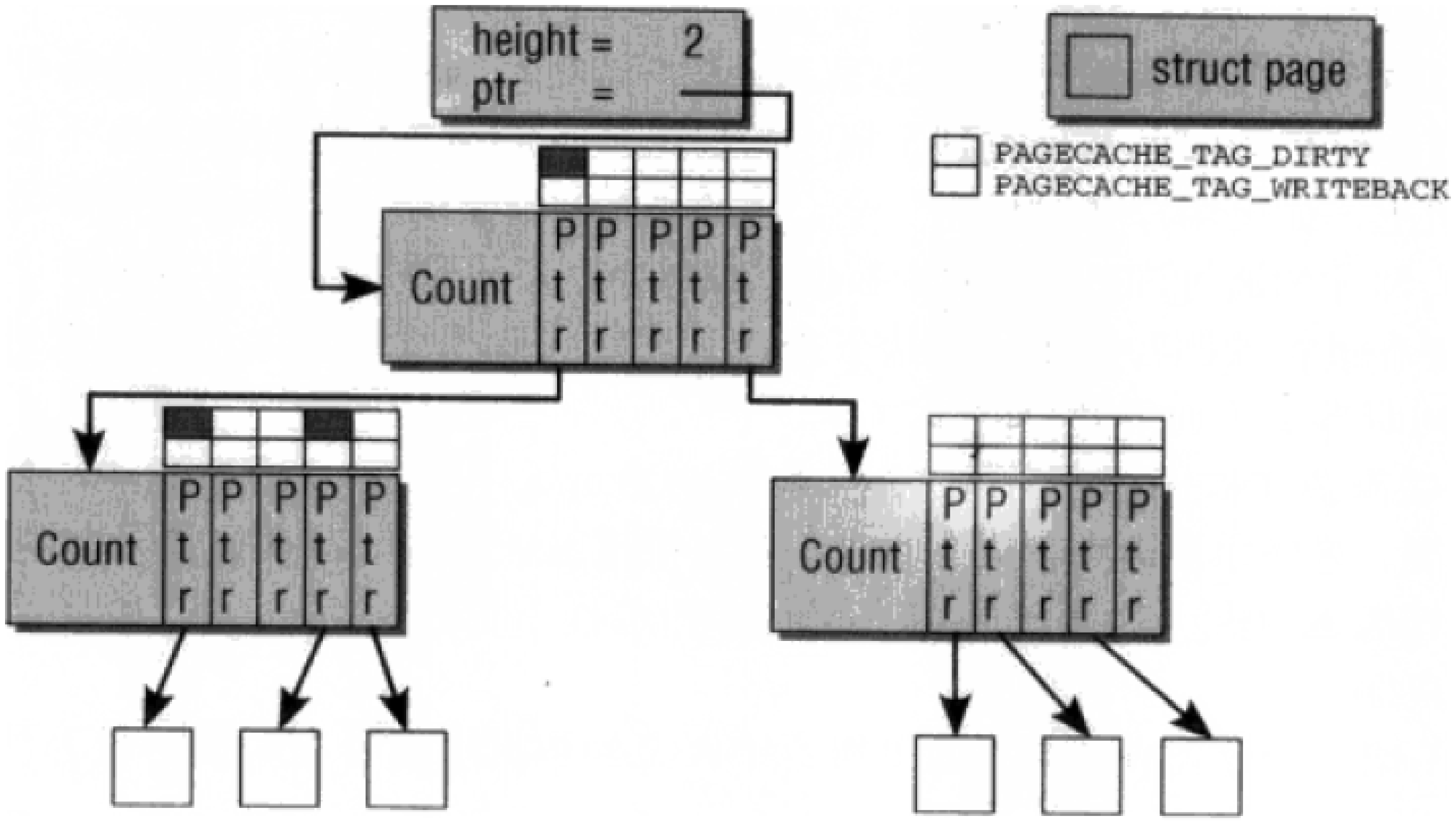
- 树由两种数据结构组成，内核中使用**void**类型来表示基数树的叶子
- **RADIX_TREE_MAP_SHIFT**决定了每个节点数组的元素数
- 节点中有两种搜索标记（**search tag**），用于标记给定是否为脏，或者是否正在回写。

linux/include/linux/radix-tree.h

```
 struct radix_tree_root {
        unsigned int   height;;
        gfp_t gfp_mask;
        struct radix_tree_node __rcu *rnode;   //根节点
};


 struct radix_tree_node {
        unsigned char   shift;          /* Bits remaining in each slot */
        unsigned char   offset;         /* Slot offset in parent */
        unsigned char   count;          /* Total entry count */
        unsigned char   exceptional;    /* Exceptional entry count */
        struct radix_tree_node *parent;         /* Used when ascending tree */
        struct radix_tree_root *root;           /* The tree we belong to */
        union {
                struct list_head private_list;  /* For tree user */
                struct rcu_head rcu_head;       /* Used when freeing node */
        };
    void __rcu      *slots[RADIX_TREE_MAP_SIZE];
    unsigned long   tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
 };
```

基数树

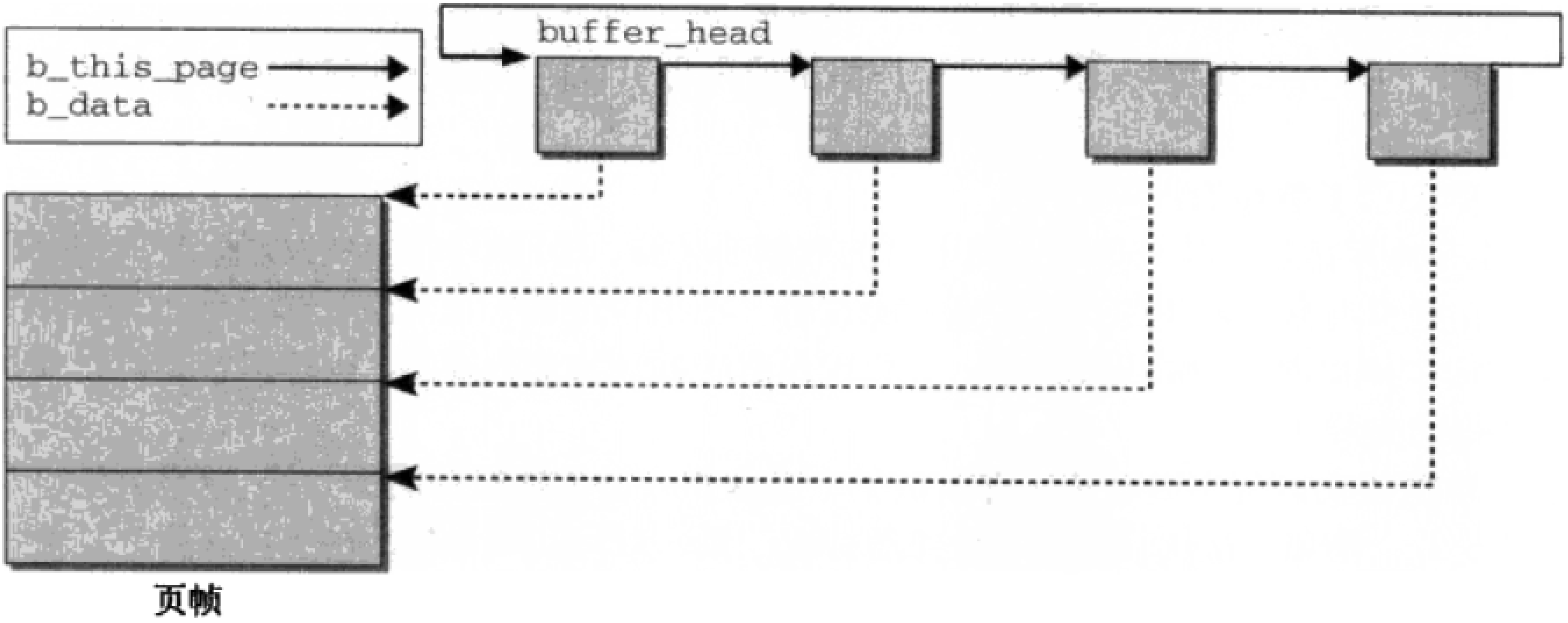


height = 2
ptr =
struct page
PAGECACHE_TAG_DIRTY
PAGECACHE_TAG_WRITEBACK
Count
Ptr
Count
Ptr
Count
Ptr

块缓存

- 块缓存的结构由两个部分组成
  1. 缓冲头
  2. 有用的数据保存在专门分配的页(可能在页缓存中)
- 独立的块缓存：独立于页缓存，缓冲头聚集在LRU管理的数组中

块缓存



地址空间

- 提供了内存中的页到后备存储器之间的映射。

```
struct address_space {
            struct inode                        *host;
            struct xarray                       i_pages;
            gfp_t                               gfp_mask;
            atomic_t                i_mmap_writable;
#ifdef CONFIG_READ_ONLY_THP_FOR_FS
            /* number of thp, only for non-shmem files */
            atomic_t                nr_thps;
#endif
            struct rb_root_cached               i_mmap;
            struct rw_semaphore                 i_mmap_rwsem;
            unsigned long                       nrpages;
            unsigned long                       nrexceptional;
            pgoff_t                             writeback_index;
            const struct address_space_operations *a_ops;
            unsigned long                       flags;
            errseq_t                wb_err;
            spinlock_t              private_lock;
            struct list_head        private_list;
            void                                *private_data;
} __attribute__((aligned(sizeof(long)))) __randomize_layout;
```

```
/**
 * struct address_space - Contents of a cacheable, mappable object.
 * @host: Owner, either the inode or the block_device.
 * @i_pages: Cached pages.
 * @gfp_mask: Memory allocation flags to use for allocating pages.
 * @i_mmap_writable: Number of VM_SHARED mappings.
 * @nr_thps: Number of THPs in the pagecache (non-shmem only).
 * @i_mmap: Tree of private and shared mappings.
 * @i_mmap_rwsem: Protects @i_mmap and @i_mmap_writable.
 * @nrpages: Number of page entries, protected by the i_pages lock.
 * @nrexceptional: Shadow or DAX entries, protected by the i_pages lock.
 * @writeback_index: Writeback starts here.
 * @a_ops: Methods.
 * @flags: Error bits and flags (AS_*).
 * @wb_err: The most recent error which has occurred.
 * @private_lock: For use by the owner of the address_space.
 * @private_list: For use by the owner of the address_space.
 * @private_data: For use by the owner of the address_space.
 */
```

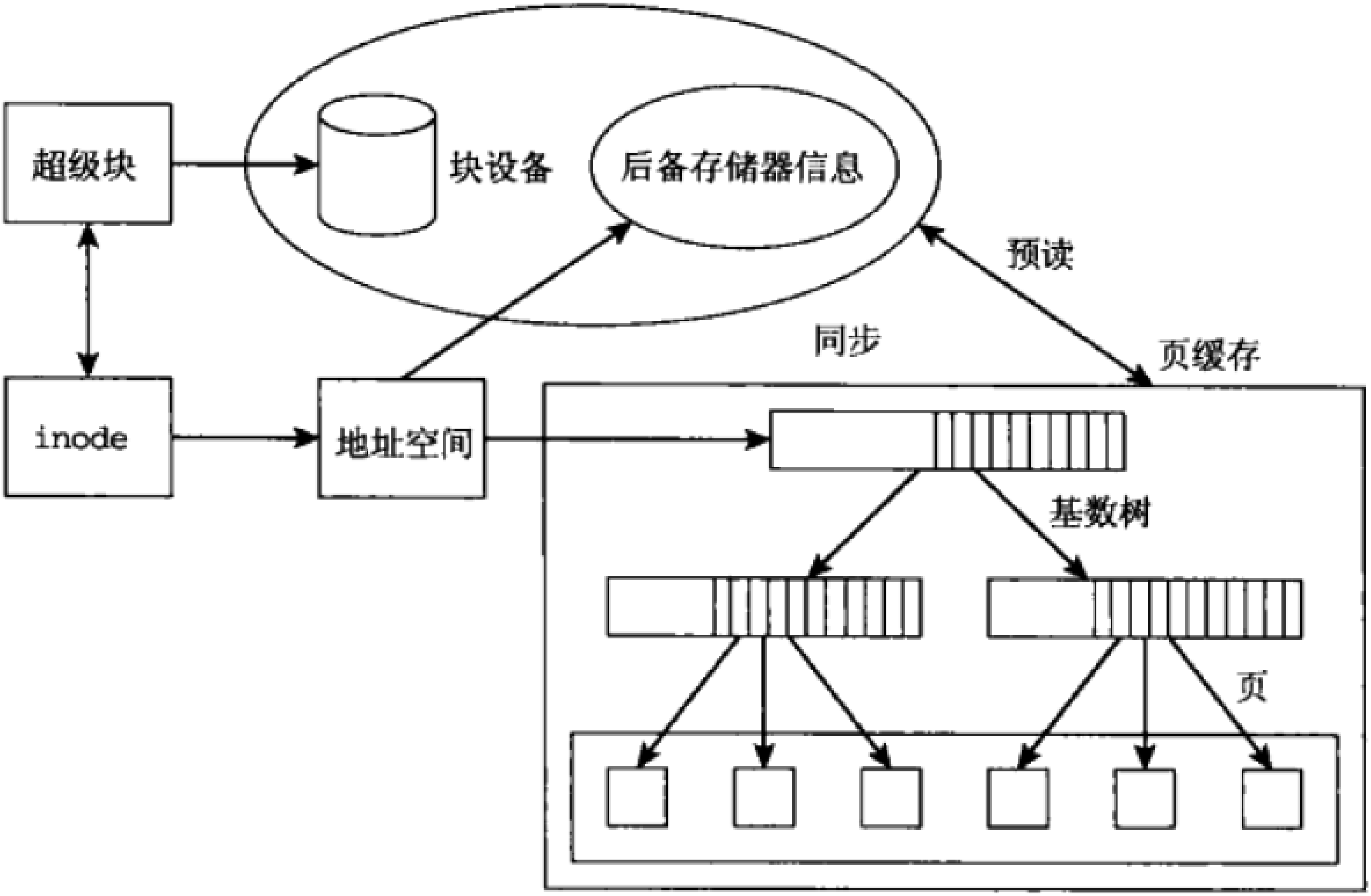
地址空间

- inode的指针指定了后备存储器
- page_tree 列出了地址空间中所有的物理内存页。
- nrpages缓存页的总数
- i_mmap包含了与inode相关的所有普通内存映射。
- backing_dev_info指向包含了与地址空间相关的后备存储器的有关信息的数据结构
- private_list 用于将buffer_head实例彼此连接起来
- flags中的标志集主要用于保存映射页所来自的GFP内存区的有关信息，也可以保存异步输入输出期间的错误信息

地址空间操作

- **Writepage**和**wirtepages**将地址空间的一页或多页写回块设备
- **Readpage**和**readpages**从后备存储器将一页或多页读入页帧
- **Sync_page**对尚未回写到后备存储器的数据进行同步
- **Set_page_dirty**将一页标记为脏
- **Prepare_wirte**和**commit_write**执行由**write**系统调用触发的写操作
- **Bmap**将地址空间内的逻辑块偏移量映射为物理块号
- **Direct_IO**用于实现直接的读写访问
- **Migrate_page**即将一页的内容移动到另外一页

地址空间



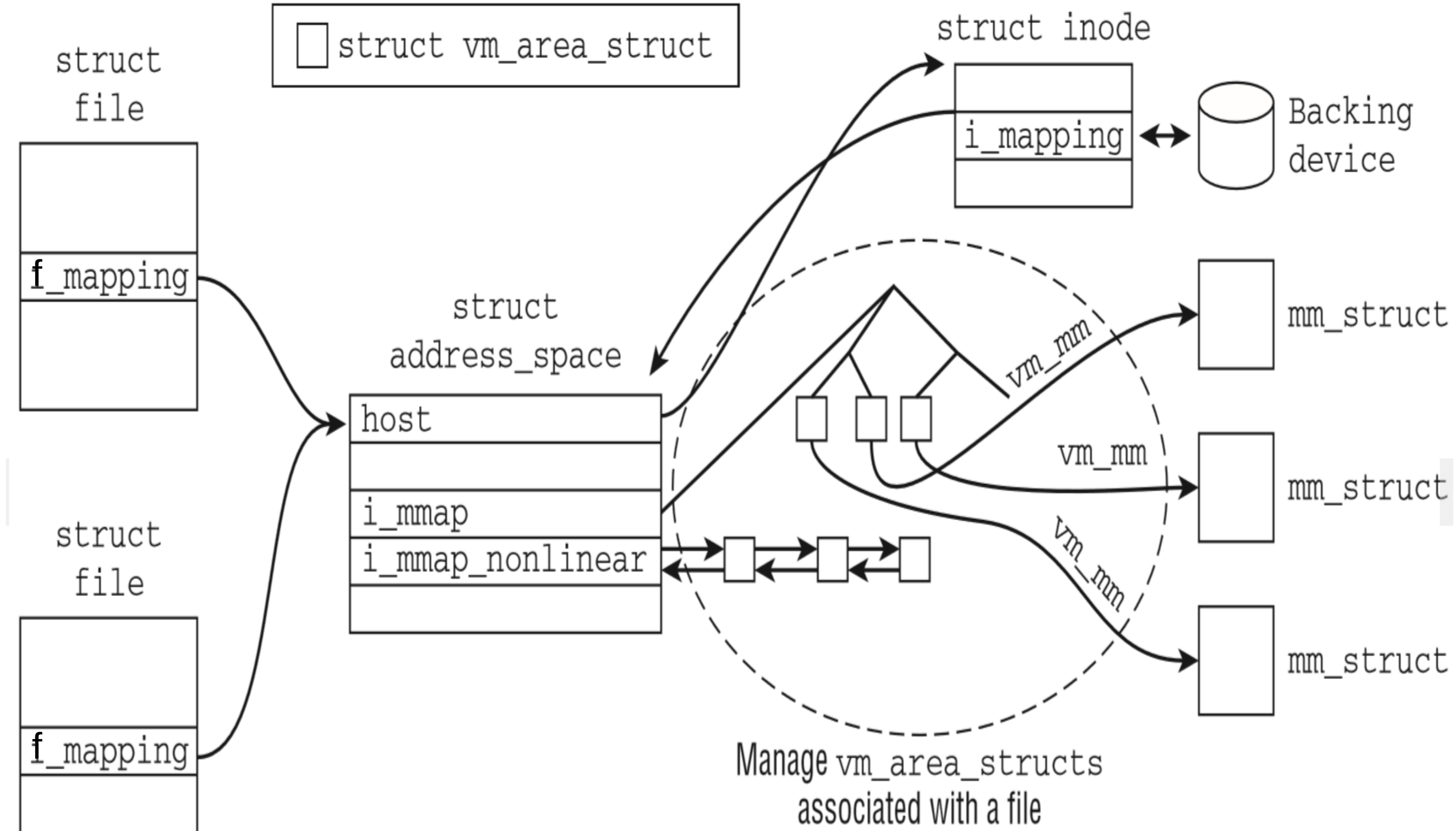
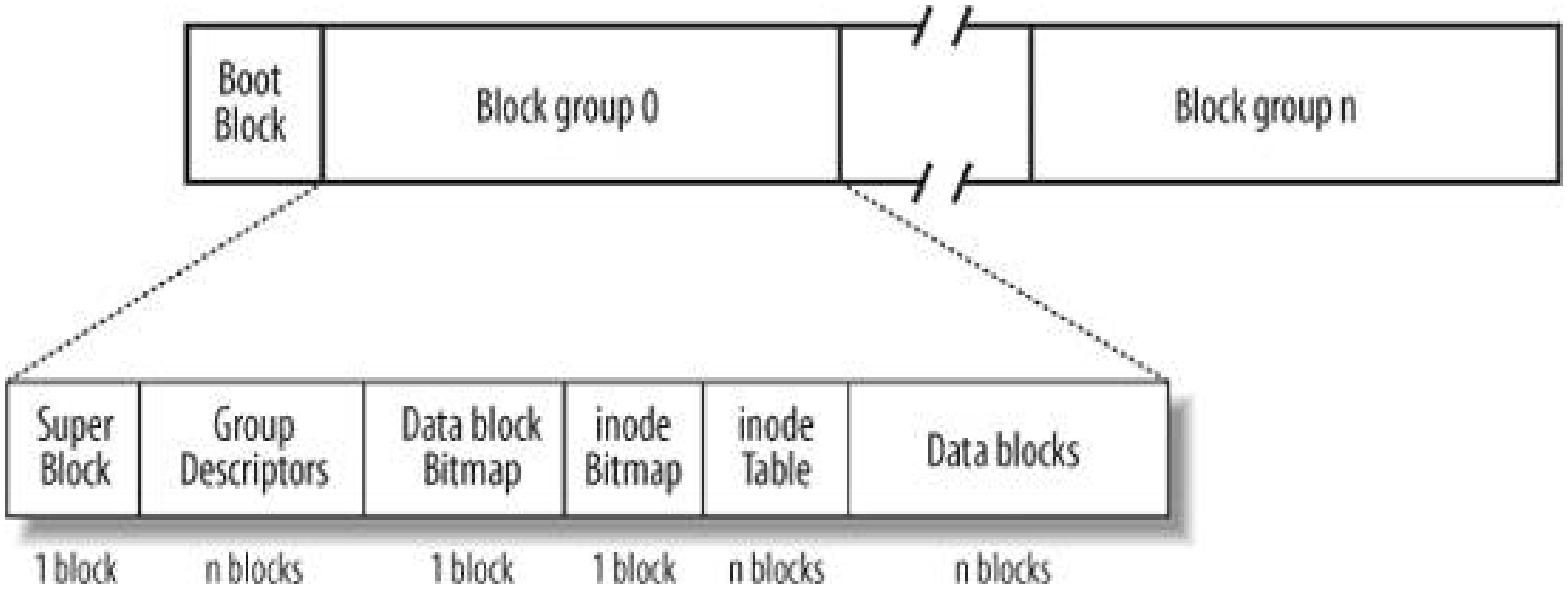


Figure 4-7: Tracking the virtual address spaces into which a given interval of a file is mapped with the help of a priority tree.

- Part 2 Ext2 文件系统
  - 和 Ext2 相关的数据结构有两种：
    - Ext2 Disk Data Structures
    - Ext2 Memory Data Structures

- Ext2 文件系统**磁盘数据结构(Disk Data Structure)**
  - 这些数据结构是物理磁盘上的数据组织的具体结构

Layouts of an Ext2 partition and of an Ext2 block group



Block Group

- superblock 和 group descriptors在每个 block group都有复制.
- 只有 block group 0 的superblock 和 group descriptors 被内核使用,其他的superblock 和 group descriptors 内核不会关心.
- Superblock 在磁盘上有多个副本,系统在 superblock 失去作用的时候,可以恢复

ext2_super_block

_ _le32	s_inodes_count	Total number of inodes
_ _le32	s_blocks_count	Filesystem size in blocks
_ _le32	s_r_blocks_count	Number of reserved blocks
_ _le32	s_free_blocks_count	Free blocks counter
_ _le32	s_free_inodes_count	Free inodes counter
_ _le32	s_first_data_block	Number of first useful block (always 1)
_ _le32	s_log_block_size	Block size
_ _le32	s_log_frag_size	Fragment size
_ _le32	s_blocks_per_group	Number of blocks per group
_ _le32	s_frags_per_group	Number of fragments per group
_ _le32	s_inodes_per_group	Number of inodes per group
_ _le32	s_mtime	Time of last mount operation
_ _le32	s_wtime	Time of last write operation
_ _le16	s_mnt_count	Mount operations counter

ext2_super_block

_ _le16	s_max_mnt_count	Number of mount operations before check
_ _le16	s_magic	Magic signature
_ _le16	s_state	Status flag
_ _le16	s_errors	Behavior when detecting errors
_ _le16	s_minor_rev_level	Minor revision level
_ _le32	s_lastcheck	Time of last check
_ _le32	s_checkinterval	Time between checks
_ _le32	s_creator_os	OS where filesystem was created
_ _le32	s_rev_level	Revision level of the filesystem
_ _le16	s_def_resuid	Default UID for reserved blocks
_ _le16	s_def_resgid	Default user group ID for reserved blocks
_ _le32	s_first_ino	Number of first nonreserved inode
_ _le16	s_inode_size	Size of on-disk inode structure
_ _le16	s_block_group_nr	Block group number of this superblock

ext2_super_block

_ _le32	s_feature_compat	Compatible features bitmap
_ _le32	s_feature_incompat	Incompatible features bitmap
_ _le32	s_feature_ro_compat	Read-only compatible features bitmap
_ _u8 [16]	s_uuid	128-bit filesystem identifier
char [16]	s_volume_name	Volume name
char [64]	s_last_mounted	Pathname of last mount point
_ _le32	s_algorithm_usage_bitmap	Used for compression
_ _u8	s_prealloc_blocks	Number of blocks to preallocate
_ _u8	s_prealloc_dir_blocks	Number of blocks to preallocate for directories
_ _u16	s_padding1	Alignment to word
_ _u32 [204]	s_reserved	Nulls to pad out 1,024 bytes

ext2_group_desc

_ _le32	bg_block_bitmap	Block number of block bitmap
_ _le32	bg_inode_bitmap	Block number of inode bitmap
_ _le32	bg_inode_table	Block number of first inode table block
_ _le16	bg_free_blocks_count	Number of free blocks in the group
_ _le16	bg_free_inodes_count	Number of free inodes in the group
_ _le16	bg_used_dirs_count	Number of directories in the group
_ _le16	bg_pad	Alignment to word
_ _le32	bg_reserved	Nulls to pad out 24 bytes

Inode Table

- The inode table consists of a series of consecutive blocks, each of which contains a predefined number of inodes
- The block number of the first block of the inode table is stored in the **bg_inode_table** field of the group descriptor.
- All inodes have the same size: **128 bytes**

Ext2 disk inode

_ _le16	**i_mode**	**File type and access rights**
_ _le16	i_uid	Owner identifier
_ _le32	i_size	File length in bytes
_ _le32	i_atime	Time of last file access
_ _le32	i_ctime	Time that inode last changed
_ _le32	i_mtime	Time that file contents last changed
_ _le32	i_dtime	Time of file deletion
_ _le16	i_gid	User group identifier
_ _le16	i_links_count	Hard links counter
_ _le32	i_blocks	Number of data blocks of the file
_ _le32	i_flags	File flags
union	osd1	Specific operating system information
_ _le32 [EXT2_N_BLOCKS]	i_block	Pointers to data blocks

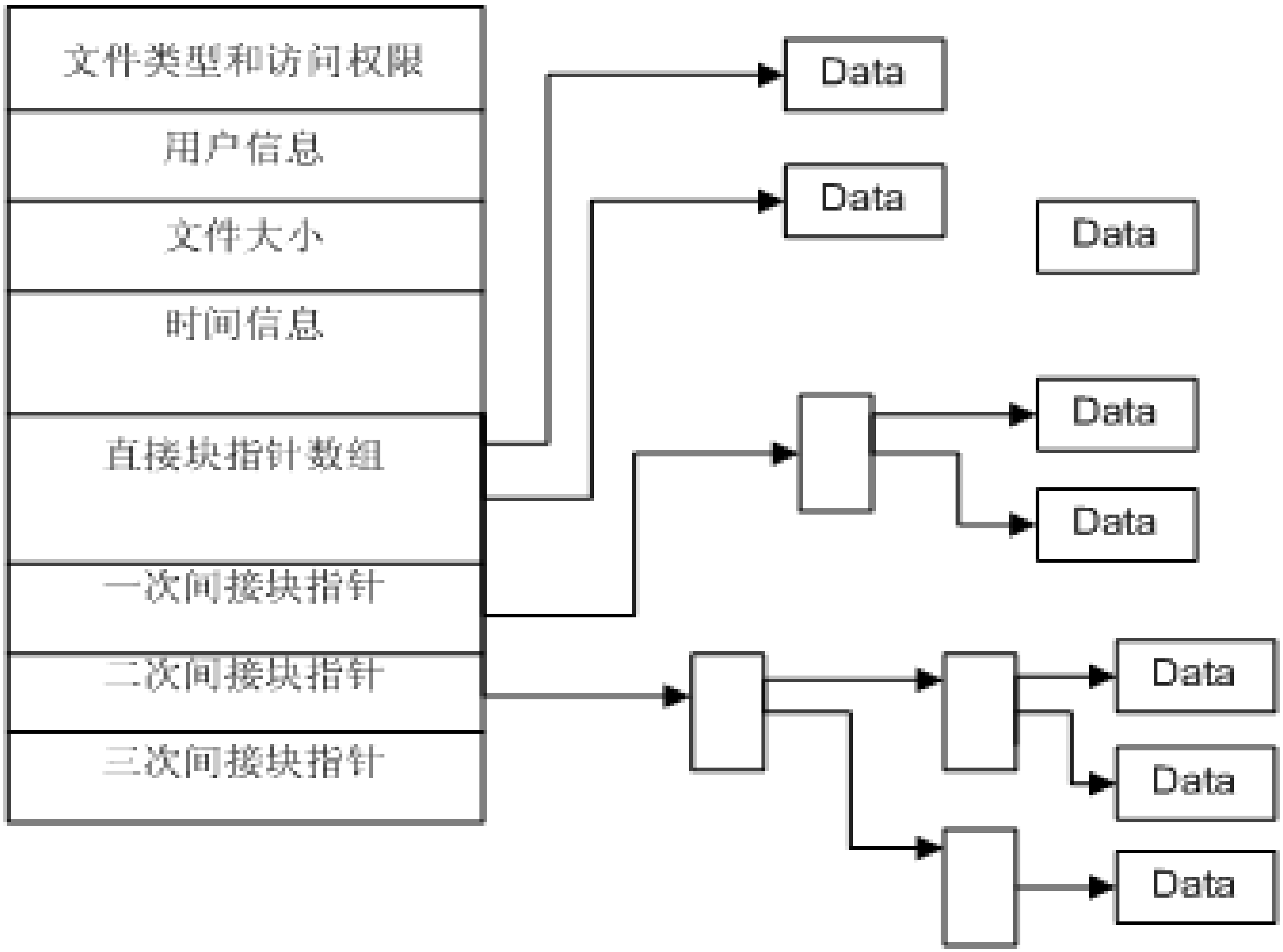
Ext2 disk inode

_ _le32	i_generation	File version (used when the file is accessed by a network filesystem)
_ _le32	i_file_acl	File access control list
_ _le32	i_dir_acl	Directory access control list
_ _le32	i_faddr	Fragment address
union	osd2	Specific operating system information

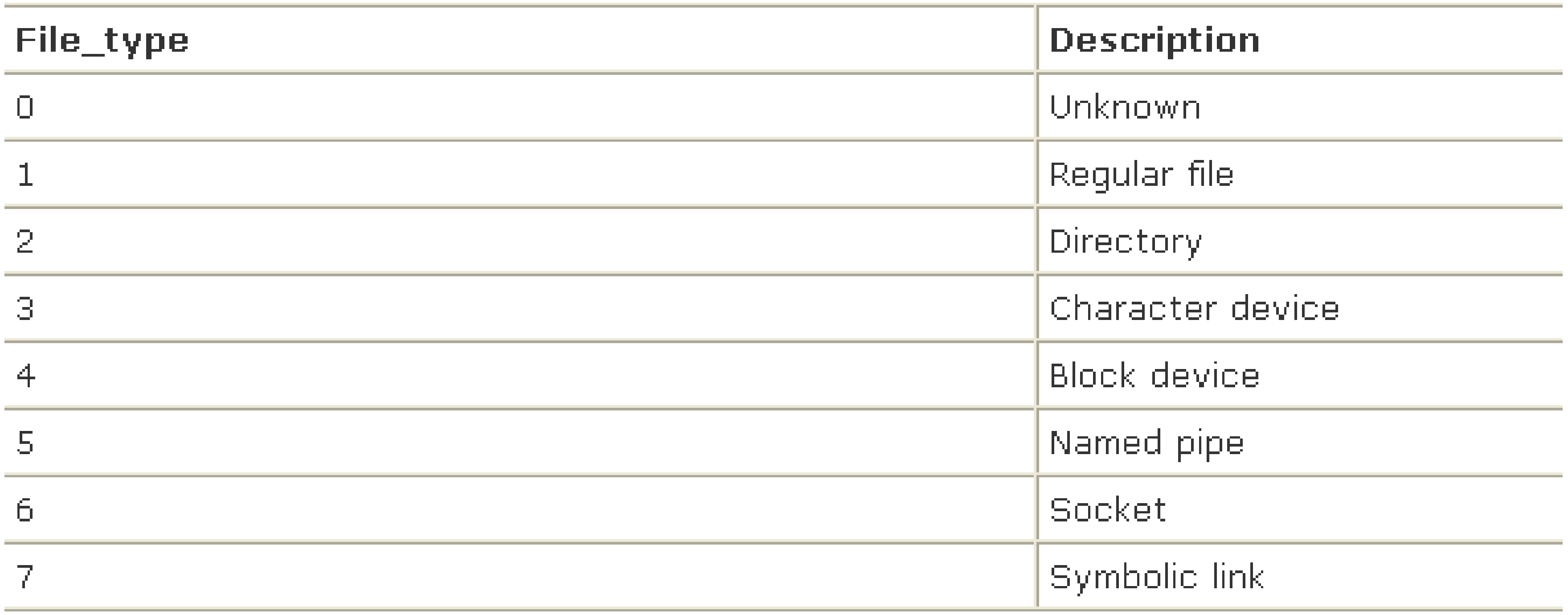
Access Control Lists

- access control list (ACL) can be associated with each file.
- chacl( ) , setfacl( ) , and getfacl( )

EXT2索引节点


文件类型和访问权限
用户信息
文件大小
时间信息
直接块指针数组
一次间接块指针
二次间接块指针
三次间接块指针
Data
Data
Data
Data
Data
Data
Data
Data

Ext2 file types



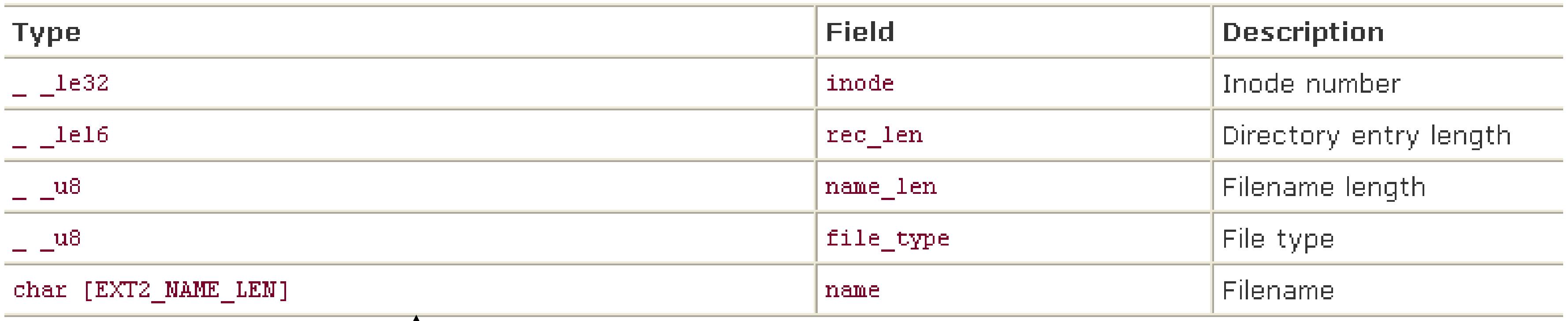
File_type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

•The different types of files recognized by Ext2 (regular files, pipes, etc.) use data blocks in different ways.

•Some files store no data and therefore need no data blocks at all

Directory

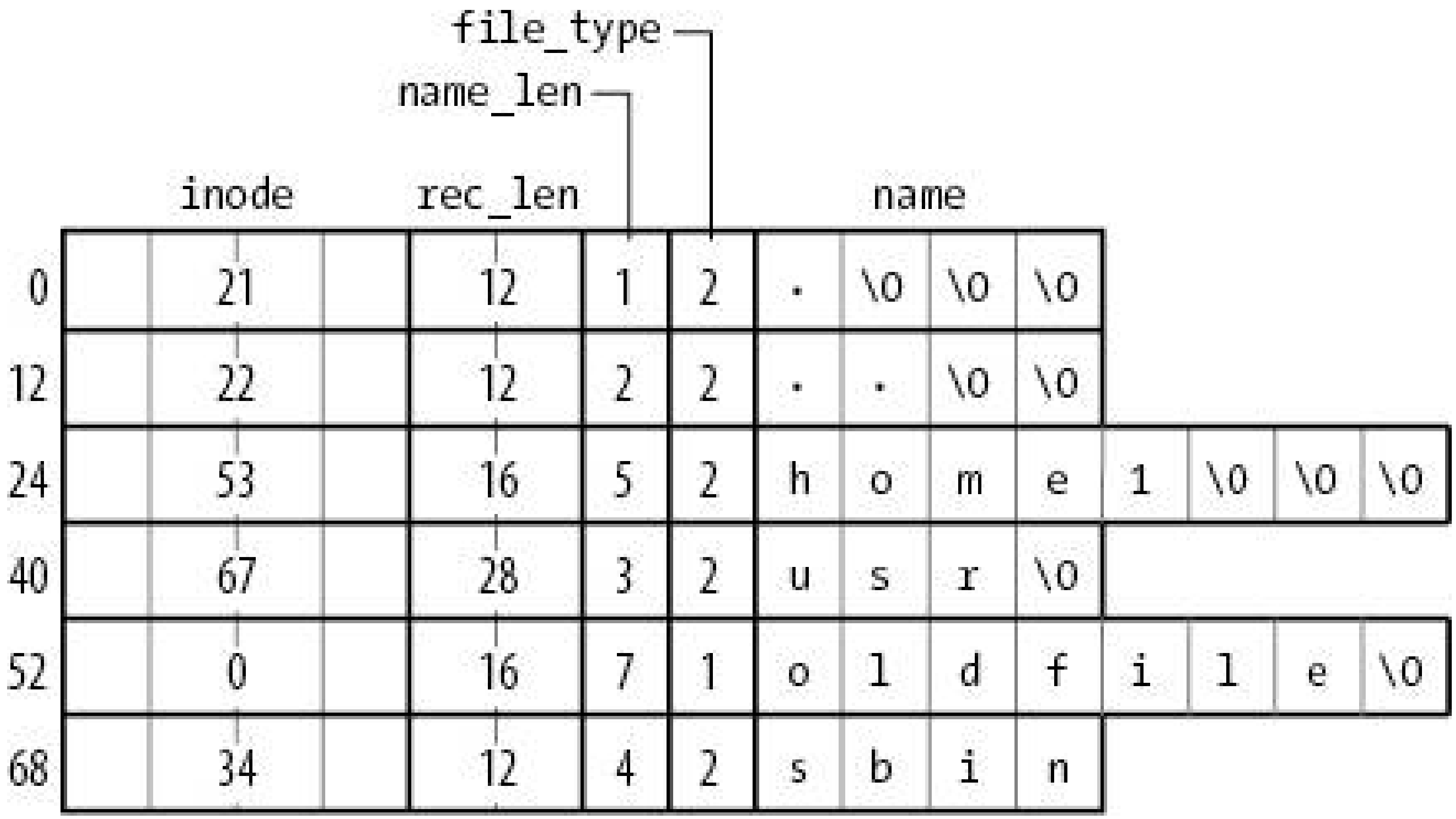
- Ext2 implements directories as a special kind of file whose **data blocks** store **filenames** together with the corresponding **inode numbers**



Type	Field	Description
_ _le32	inode	Inode number
_ _le16	rec_len	Directory entry length
_ _u8	name_len	Filename length
_ _u8	file_type	File type
char [EXT2_NAME_LEN]	name	Filename

The fields of an Ext2 directory entry

An example of the Ext2 directory



Symbolic link

- if the pathname of a symbolic link has up to 60 characters, it is stored in the **i_block** field of the inode, which consists of an array of 15 4-byte integers; no data block is therefore required.
- If the pathname is longer than 60 characters, however, a single data block is required.

Device file, pipe, and socket

- No data blocks are required for these kinds of files. All the necessary information is stored in the inode.

- Ext2 Memory Data Structures

Ext2 的磁盘数据结构常驻内存

- 为了提高效率, Ext2 文件系统的磁盘数据结构会在被 **mount** 的时候拷贝到内存中,因此,以后就减少了读磁盘次数,提高了效率

常用的磁盘数据结构的举例

- 当新文件创建的时候，superblock 结构的 s_free_inodes_count 域以及相应的group descriptor 的域 bg_free_inodes_count 的数值减少
- 内核往一个文件中追加数据（为其分配的data block数增加），Ext2 superblock 的 s_free_blocks_count 域和 group descriptor 的bg_free_blocks_count 域都要被修改
- 重写存在的文件的某个部分需要修改ext2 superblock 的 s_wtime 域（最后一次写操作的时间）.

Disk data structure and memory data structure

Type	Disk data structure	Memory data structure	Caching mode
Superblock	ext2_super_block	ext2_sb_info	Always cached
Group descriptor	ext2_group_desc	ext2_group_desc	Always cached
Block bitmap	Bit array in block	Bit array in buffer	Dynamic
inode bitmap	Bit array in block	Bit array in buffer	Dynamic
inode	ext2_inode	ext2_inode_info	Dynamic
Data block	Array of bytes	VFS buffer	Dynamic
Free inode	ext2_inode	None	None
Free block	Array of bytes	None	Never

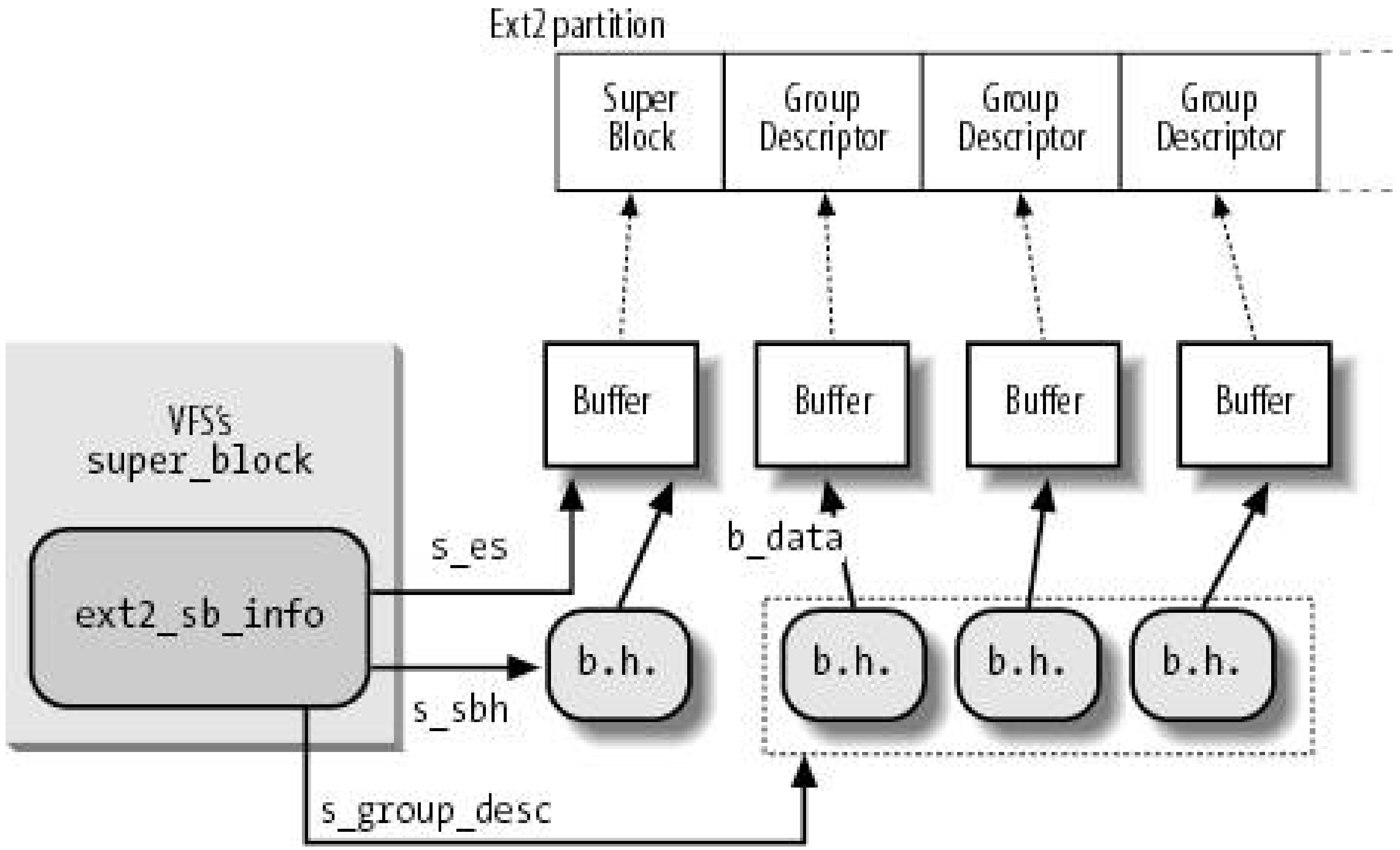
- The never-cached data is not kept in any cache because it does not represent meaningful information
- the always-cached data is always present in RAM (**keeping the page's usage counter > 0**), thus it is never necessary to read the data from disk (periodically, however, the data must be written back to disk).
- the dynamically cached data is kept in a cache as long as the associated object is in use; when the file is closed or the data block is deleted, the page frame reclaiming algorithm may remove the associated data from the cache.

The Ext2 Superblock Object

- **VFS superblock's** s_fs_info field points to a structure containing filesystem-specific data. In the case of Ext2, this field points to a structure of type ext2_sb_info,it contains:
  - Most of the disk superblock fields
  - An s_sbh pointer to the buffer head of the buffer containing the disk superblock
  - An s_es pointer to the buffer containing the disk superblock
  - An s_group_desc pointer to an array of buffer heads of buffers containing the group descriptors (usually, a single entry is sufficient)
  - Other data related to mount state, mount options, and so on

a buffer page is a page of data associated with additional descriptors called "buffer heads ," whose main purpose is to quickly locate the disk address of each individual block in the page

The ext2_sb_info data structure



ext2_fill_super( )

- When the kernel **mounts** an Ext2 filesystem, it invokes the ext2_fill_super( ) function to allocate space for the data structures and to fill them with data read from disk
  - Allocates an **ext2_sb_info descriptor** and stores its address in the s_fs_info field of the VFS's superblock object.
  - Invokes **_ _bread( )** to **allocate** a buffer in a buffer page together with the corresponding buffer head, and to **read** the superblock from disk into the buffer

ext2_fill_super( )

- Allocates an array of bytes, one byte for each group and stores its address in the **s_debts** field of the ext2_sb_info descriptor
- Allocates an array of pointers to buffer heads, one for each group descriptor, and stores the address of the array in the s_group_desc field of the ext2_sb_info descriptor.
- Invokes repeatedly _ _bread( ) to allocate buffers and to read from disk the blocks containing the Ext2 group descriptors; stores the addresses of the buffer heads in the s_group_desc array allocated in the previous step.
- Allocates an inode and a dentry object for the root directory, and sets up a few fields of the superblock object so that it will be possible to read the root inode from disk.

The Ext2 Superblock Object

- All the data structures allocated by ext2_fill_super( ) are kept in memory after the function returns;
- they will be released only when the Ext2 filesystem will be **unmounted**.
- When the kernel must **modify** a field in the Ext2 superblock, it simply writes the new value in the proper position of the corresponding **buffer** and then marks the buffer as **dirty**.

The Ext2 inode Object

- When opening a file, a pathname lookup is performed. For each component of the pathname that is not already in the dentry cache , **a new dentry object and a new inode object are created**
- **When the VFS accesses an Ext2 disk inode, it creates a corresponding inode descriptor of type ext2_inode_info**

ext2_inode_info

- The whole VFS inode object stored in the field vfs_inode
- Most of the fields found in the disk's inode structure that **are not kept** in the VFS inode
  - i_block_group: block group index at which the inode belongs
  - i_next_alloc_block and i_next_alloc_goal : store the logical block number and the physical block number of the disk block that was most recently allocated to the file, respectively
  - The i_prealloc_block and i_prealloc_count fields, which are used for data block preallocation
  - The xattr_sem field, a read/write semaphore that allows extended attributes to be read concurrently with the file data
  - The i_acl and i_default_acl fields, which point to the ACLs of the file

Creating the Ext2 Filesystem

- *mke2fs*命令
  - Block size: 1,024 bytes (default value for a small filesystem)
  - Fragment size: block size (block fragmentation is not implemented)
  - Number of allocated inodes: 1 inode for each 8,192 bytes
  - Percentage of reserved blocks: 5 percent

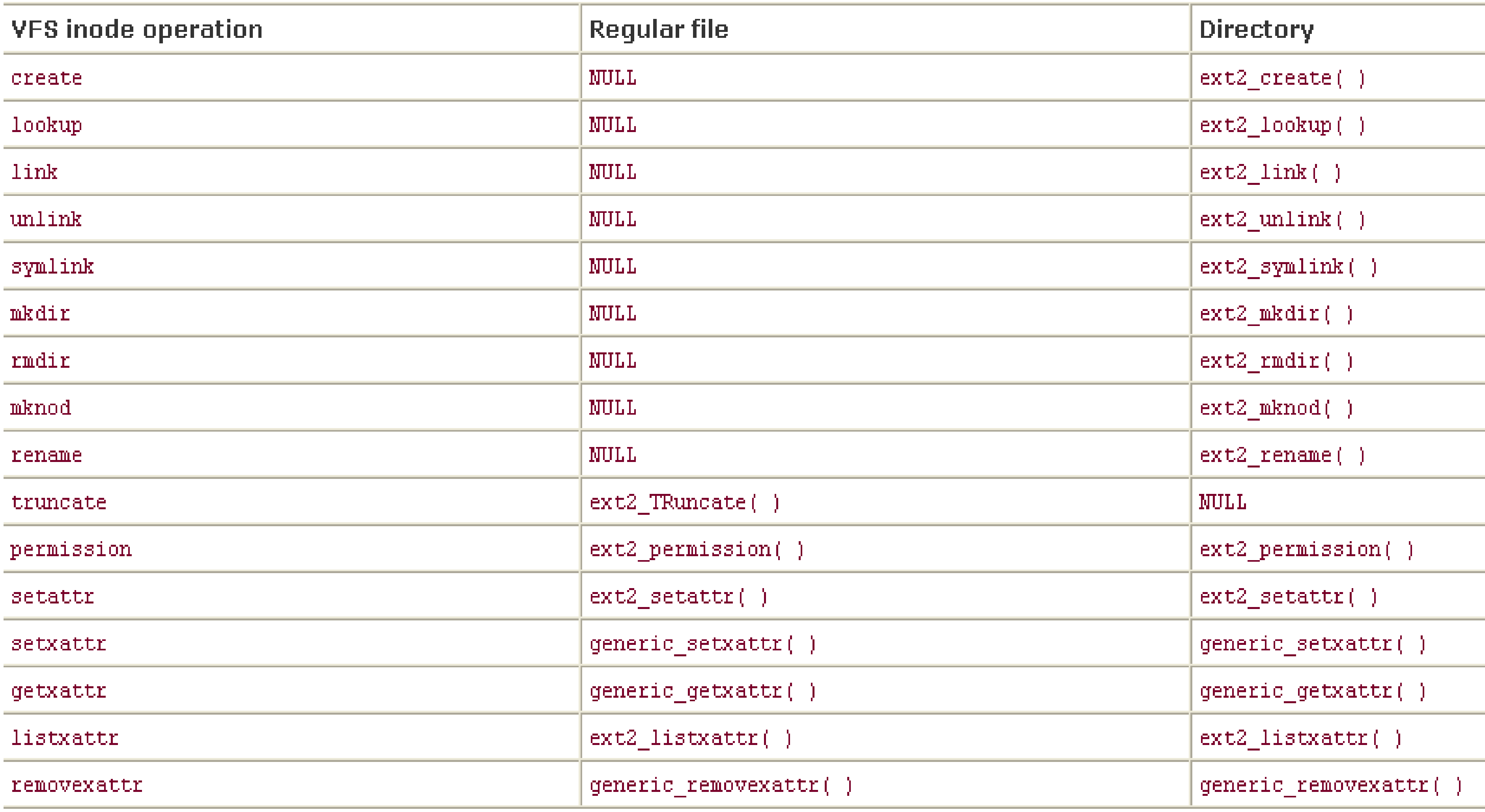
mke2fs

- Initializes the superblock and the group descriptors.
- Optionally, checks whether the partition contains defective blocks; if so, it creates a list of defective blocks.
- For each block group, reserves all the disk blocks needed to store the superblock, the group descriptors, the inode table, and the two bitmaps.
- Initializes the inode bitmap and the data block bitmap of each block group to 0.
- Initializes the inode table of each block group.
- Creates the */root* directory.
- Creates the *lost+found* directory, which is used by *e2fsck* to link the lost and found defective blocks.
- Updates the inode bitmap and the data block bitmap of the block group in which the two previous directories have been created.
- Groups the defective blocks (if any) in the *lost+found* directory.

Ext2 Superblock Operations

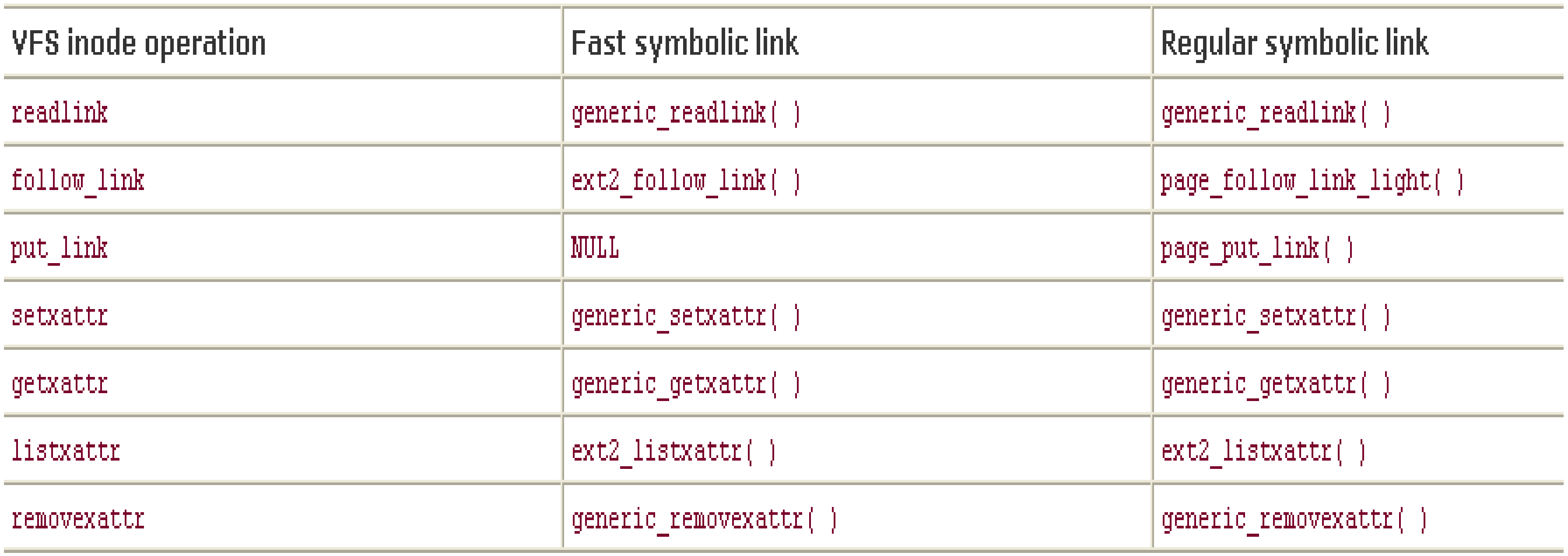
- Many VFS superblock operations have a specific implementation in Ext2, namely alloc_inode, destroy_inode, read_inode, write_inode, delete_inode, put_super, write_super, statfs, remount_fs, and clear_inode
- The addresses of the superblock methods are stored in the **ext2_sops** array of pointers

Ext2 inode Operations



VFS inode operation	Regular file	Directory
create	NULL	ext2_create( )
lookup	NULL	ext2_lookup( )
link	NULL	ext2_link( )
unlink	NULL	ext2_unlink( )
symlink	NULL	ext2_symlink( )
mkdir	NULL	ext2_mkdir( )
rmdir	NULL	ext2_rmdir( )
mknod	NULL	ext2_mknod( )
rename	NULL	ext2_rename( )
truncate	ext2_TRuncate( )	NULL
permission	ext2_permission( )	ext2_permission( )
setattr	ext2_setattr( )	ext2_setattr( )
setxattr	generic_setxattr( )	generic_setxattr( )
getxattr	generic_getxattr( )	generic_getxattr( )
listxattr	ext2_listxattr( )	ext2_listxattr( )
removexattr	generic_removexattr( )	generic_removexattr( )

Ext2 inode operations for fast and regular symbolic links



VFS inode operation	Fast symbolic link	Regular symbolic link
readlink	generic_readlink( )	generic_readlink( )
follow_link	ext2_follow_link( )	page_follow_link_light( )
put_link	NULL	page_put_link( )
setxattr	generic_setxattr( )	generic_setxattr( )
getxattr	generic_getxattr( )	generic_getxattr( )
listxattr	ext2_listxattr( )	ext2_listxattr( )
removexattr	generic_removexattr( )	generic_removexattr( )

Ext2 File Operations

VFS file operation	Ext2 method
llseek	generic_file_llseek( )
read	generic_file_read( )
write	generic_file_write( )
aio_read	generic_file_aio_read( )
aio_write	generic_file_aio_write( )
ioctl	ext2_ioctl( )
mmap	generic_file_mmap( )
open	generic_file_open( )
release	ext2_release_file( )
fsync	ext2_sync_file( )
readv	generic_file_readv( )
writev	generic_file_writev( )
sendfile	generic_file_sendfile( )

ext2_fill_super() → ext2_iget() → ext2_set_file_ops()

```
void ext2_set_file_ops(struct inode *inode)
{
        inode->i_op = &ext2_file_inode_operations;
        inode->i_fop = &ext2_file_operations;
        if (IS_DAX(inode))
                inode->i_mapping->a_ops = &ext2_dax_aops;
        else if (test_opt(inode->i_sb, NOBH))
                inode->i_mapping->a_ops = &ext2_nobh_aops;
        else
                inode->i_mapping->a_ops = &ext2_aops;
}
```

Managing Ext2 Disk Space

- Creating inodes
- Deleting inodes

Creating inodes

- Ext2_new_inode()创建一个Ext2 disk inode，返回一个相应的 inode object.
- 1、调用 new_inode( ) 分配一个新的 **VFS inode object**; 初始化它的 i_sb 域为 superblock 地址，这个地址存放在 dir->i_sb；把它添加到 in-use inode list 以及 superblock's list
- 2、如果新的inode 是**目录**，那么调用 find_group_orlov() ，**找到一个合适的 block group**.
  - A.如果创建目录的父目录是根目录，那么查找block group 的工作是在所有的 block group 中找，找到一个其 free blocks 以及 free inodes 大于平均水平的 group;如果找不到这样的组，那么转 C
  - B.创建的目录的父目录不是根目录，如果父目录所在组满足如下条件，那么把新创建的目录 inode 放到父目录所在的 group：
    - 组没有包含太多的目录
    - 组有足够的 free inode
    - **组有很小的 “debt”(创建新目录，那么debt增加；创建其他文件，减小)**

Creating inodes

  - C、如果没有好的组发现，那么从包含父目录的 group 开始寻找 free inodes 大于平均水平的第一组
- 3、如果创建的inode**不是目录**，那么调用find_group_other() 来在一个**拥有 free inode的组**中分配
  - 搜索方法：从包括父目录的group开始进行logarithmic搜索。举例来说，父目录所在group号是i，n是总的group的数目，那么下一步搜索 i mod (n), 然后是 i +1 mod (n)，i+1+2 mod (n), i+1+2+4 mod (n)
  - 如果以上方法不能得到合适的组，那么从包含父目录的组开始线性搜索

Creating inodes

- 4、调用 read_inode_bitmap()获得选中的group 的 **inode bitmap**，获得第一个 null bit，然后得到第一个 free disk inode 的号
- 5、分配disk inode: 设置inode bitmap 中的相应的位，然后标记包含bitmap的缓冲区为 **dirty**. 如果文件系统是被 mounted，而且设置了 MS_SYNCHRONOUS 标志，那么这个函数还要调用 sync_dirty_buffer( ) 开始 I/O 写操作，直到操作结束.

Creating inodes

- 6、减少 group descriptor 的 bg_free_inodes_**count** 域的值；如果新的 inode 是目录，那么函数增加 bg_used_dirs_**count** 域的值，然后标志包含 group descriptor 的缓冲区为**dirty**.
- 7、根据 inode 是否是普通文件，还是目录增加或减少 superblock的 **s_debts**数组中的group's counter

Creating inodes

- 8、减小 ext2_sb_info 数据结构的 s_freeinodes_**counter** 域; 如果新的 inode 是目录，减小 ext2_sb_info 的 **s_dirs_counter** 域.
- 9、设置superblock的 s_dirt 标志为1, 然后标记包含superblock 的buffer 为 **dirty**.
- 10、设置VFS’s 的 superblock object的 s_**dirt** 域为 1.

Creating inodes

- 11、初始化 inode object.设置inode number i_no 并拷贝 xtime.tv_sec 的值到**i_atime, i_mtime, 和 i_ctime**.
- 12、初始化 inode的 **ACLs**.
- 13、把新的 inode 插入到 hash table **inode_hashtable**，然后调用mark_inode_dirty( ) 移动 inode object 到 superblock's dirty inode list
- 14、调用 ext2_preread_inode( ) 从磁盘上读取包含 inode的 block，然后把 block 放到 **page cache** 中.
- 15、**返回新的inode object 地址**.

Deleting inodes

- ext2_free_inode( ) 删除一个 disk inode
- 1、调用 clear_inode( ),它调用如下操作：
  - Removes any dirty "indirect" buffer associated with the inode
  - If the I_LOCK flag of the inode is set, some of the inode's buffers are involved in I/O data transfers; the function suspends the current process until these I/O data transfers terminate.
  - Invokes the clear_inode method of the superblock object, if defined
  - If the inode refers to a device file, it removes the inode object from the device's list of inodes; this list is rooted either in the list field of the cdev character device descriptor
  - Sets the state of the inode to I_CLEAR (the inode object contents are no longer meaningful).

Deleting inodes

- 2、Computes the index of the block group containing the disk inode from the inode number and the number of inodes in each block group.
- 3、调用 read_inode_bitmap( ) 得到 inode bitmap.
- 4、增加 group descriptor 的 bg_free_inodes_count 域. 如果删除的inode 是目录，那么它减小bg_used_dirs_count 域.设置包含group descriptor 的 buffer 为 dirty.

Deleting inodes

- 5、如果删除的 inode 是目录，它减小 ext2_sb_info 数据结构的 s_dirs_counter 域，设置superblock 的 s_dirt 为1, 然后标志包括它的 buffer 为dirty.
- 6、Clears the bit corresponding to the disk inode in the inode bitmap and marks the buffer that contains the bitmap as dirty. Moreover, if the filesystem has been mounted with the MS_SYNCHRONIZE flag, it invokes sync_dirty_buffer( ) to wait until the write operation on the bitmap's buffer terminates.

- Part 3 Ext3 & Ext4 文件系统
  - – Ext3文件系统
  - – Ext4文件系统

The Ext3 Filesystem

- a journaling filesystem
- To be, as much as possible, compatible with the old Ext2 filesystem

Ext2 的一些问题

- 对于文件系统的修改，最终的可能是数据要放到缓存一段时间才能写入硬盘，这样的话如果系统出现一些致命的问题，比如断电，硬盘上的数据就会可能出现不一致的现象，而且检测这种不一致需要花费很长的时间。
- The goal of a **journaling filesystem** is to **avoid running time-consuming consistency checks** on the whole filesystem by looking instead in a **special disk area that contains the most recent disk write operations named journal.** Remounting a journaling filesystem after a system failure is a matter of a few seconds.

The Ext3 Journaling Filesystem

- The idea behind Ext3 journaling is to perform each high-level change to the filesystem in **two steps.**
  - First, a copy of the blocks to be written is **stored in the journal**; then, when the I/O data transfer to the journal is completed (in short, data is committed to the journal), the blocks are **written in the filesystem**.
  - When the I/O data transfer to the filesystem terminates (data is committed to the filesystem), the copies of the blocks in the journal are discarded.

The Ext3 Journaling Filesystem

- While **recovering** after a system failure, the ***e2fsck*** program distinguishes the following two cases:
  - The system failure occurred **before a commit** to the journal
    - Either the copies of the blocks relative to the high-level change are missing from the journal or they are incomplete; in both cases, *e2fsck* ignores them.
  - The system failure occurred **after a commit** to the journal
    - The copies of the blocks are valid, and *e2fsck* writes them into the filesystem.

The Ext3’s Metadata

- 有 6 种 metadata: superblocks, group block descriptors, inodes, blocks used for indirect addressing (indirection blocks), data bitmap blocks, and inode bitmap blocks.
- 其他文件系统可能使用不同的 metadata.

Ext3 's three different journaling modes

- Journal
  - **All filesystem data and metadata changes are logged into the journal**. This mode minimizes the chance of losing the updates made to each file, but it requires many additional disk accesses. For example, when a new file is created, all its data blocks must be duplicated as log records. This is the safest and slowest Ext3 journaling mode.
- Ordered
  - **Only changes to filesystem metadata are logged into the journal.** However, the Ext3 filesystem groups metadata and relative data blocks so that data blocks are written to disk before the metadata. This way, the chance to have data corruption inside the files is reduced; for instance, each write access that enlarges a file is guaranteed to be fully protected by the journal.
- Writeback
  - **Only changes to filesystem metadata are logged**; this is the method found on the other journaling filesystems and is the fastest mode.

The Ext4 Filesystem

- Ext4 is part of the Linux 2.6.28 kernel
- Ext4 is the evolution of the most used Linux filesystem, Ext3
- Ext4 is a deeper improvement over Ext3 than Ext3 was over Ext2
  - Ext3 was mostly about adding journaling to Ext2
  - Ext4 modifies important data structures of the filesystem → better performance, reliability and features

Compatibility

- Any existing Ext3 filesystem can be migrated to Ext4 with an easy procedure which consists in running a couple of commands in read-only mode
- The procedure is safe and doesn't risk your data (obviously, backup of critical data is recommended, even if you aren't updating your filesystem :).
- Ext4 will use the new data structures only on new data

Bigger filesystem/file sizes

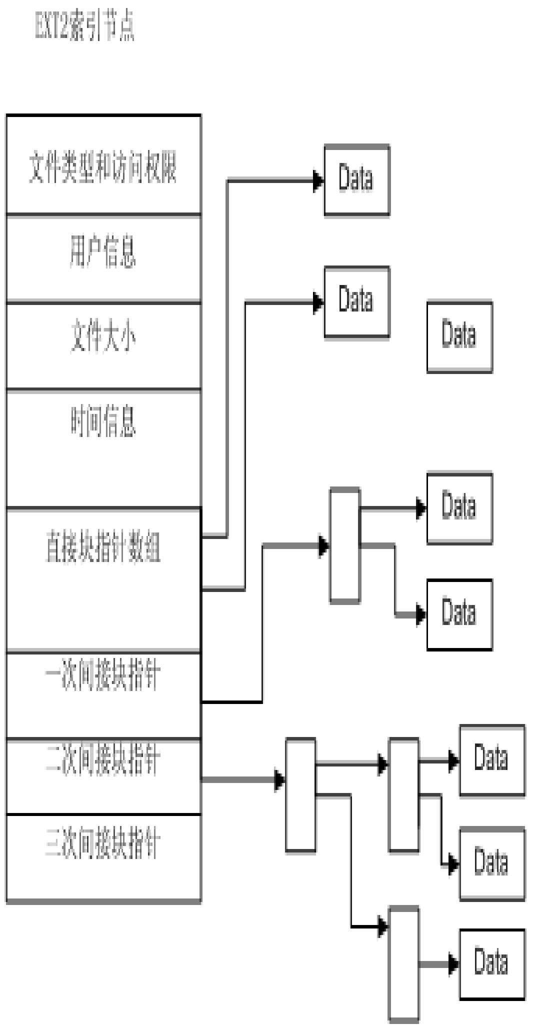
- Ext3 support 16 TB of maximum filesystem size, and 2 TB of maximum file size.
- Ext4 adds **48-bit block addressing**, so it will have 1 EB of maximum filesystem size and 16 TB of maximum file size.
  - 1 EB = 1,048,576 TB (1 EB = 1024 PB, 1 PB = 1024 TB, 1 TB = 1024 GB).
  - Why 48-bit and not 64-bit? There are some limitations that would need to be fixed before making Ext4 fully 64-bit capable.

Sub directory scalability

- the maximum possible number of sub directories contained in a single directory in Ext3 is 32000.
- Ext4 breaks that limit and allows a unlimited number of sub directories.

Extents

- The traditionally Unix-derived filesystems like Ext3 use indirect block mapping scheme
- Modern filesystems use "extents"
  - An extent is basically a bunch of **contiguous** physical blocks. It basically says "The data is in the next n blocks".
    - For example, a 100 MB file can be allocated into a single extent of that size, instead of needing to create the indirect mapping for 25600 blocks (4 KB per block).
  - Huge files are split in several extents.
  - Extents improve the performance and also help to reduce the fragmentation, since an extent encourages continuous layouts on the disk.



Multiblock allocation

- When need to write new data to the disk, there's a **block allocator** that decides which free blocks will be used to write the data.
- Ext3 block allocator only allocates one block (4KB) at a time.
  - if the system needs to write the 100 MB data, it will need to call the block allocator 25600
  - Inefficient
  - No location policy optimization because it doesn't knows total data being allocated
- **Ext4** uses a "**multiblock allocator**" (**mballoc**) which allocates many blocks in a single call
  - Improves the performance
  - particularly useful with delayed allocation and extents

Delayed allocation

- delaying the allocation of blocks as much as possible
- a performance feature (doesn't change the disk format)
- traditionally filesystems (such as Ext3, reiser3, etc) : allocate the blocks as soon as possible.
  - if a process write()s, the filesystem code will allocate immediately the blocks where the data will be placed - even if the data is not being written right now to the disk and it's going to be kept in the cache for some time.
- Delayed allocation, does not allocate the blocks immediately when the process write()s, rather, it delays the allocation of the blocks while the file is kept in cache, until it is really going to be written to the disk.
  - This gives the block allocator the opportunity to optimize the allocation.
- Delayed allocation plays very nicely with the two previous features mentioned, extents and multiblock allocation
  - in many workloads when the file is written finally to the disk it will be allocated in extents whose block allocation is done with the mballoc allocator. The performance is much better, and the fragmentation is much improved in some workloads.

Fast fsck

- Fsck is a very slow operation, especially the first step: checking all the inodes in the file system.
- In **Ext4**, at the end of each group's inode table will be stored a list of unused inodes (with a checksum, for safety), so fsck will not check those inodes. The result is that total fsck time improves from 2 to 20 times
- Notice: fsck builds the list of unused inodes
  - This means that you must run fsck to get the list of unused inodes built, and only the next fsck run will be faster (you need to pass a fsck in order to convert a Ext3 filesystem to **Ext4**)

Journal checksumming

- Ext4 checksums the journal data to know if the journal blocks are failing or corrupted.
- journal checksumming has a bonus: it allows one to convert the two-phase commit system of Ext3's journaling to a single phase, speeding the filesystem operation up to 20% in some cases - so reliability and performance are improved at the same time.

"No Journaling" mode

Inode-related features

- Larger inodes, nanosecond timestamps, fast extended attributes, inodes reservation...
- Larger inodes:
  - Ext3 supports configurable inode sizes (via the -I mkfs parameter), but the default inode size is 128 bytes.
  - Ext4 default to 256 bytes.
    - accommodate some extra fields (like nanosecond timestamps or inode versioning)
    - store extend attributes
      - This will make the access to those attributes much faster, and improves the performance of applications that use extend attributes by a factor of 3-7 times.

Inode-related features

Inode reservation

- reserving several inodes when a directory is created, expecting that they will be used in the future.
- This improves the performance, because when new files are created in that directory they'll be able to use the reserved inodes. File creation and deletion is hence more efficient.

Inode-related features

Nanoseconds timestamps

- inode fields like "modified time" will be able to use nanosecond resolution instead of the second resolution of Ext3.

Persistent preallocation

- available in Ext3 in the latest kernel versions, and emulated by glibc in the filesystems that don't support it
- allows applications to preallocate disk space:
  - Applications tell the filesystem to preallocate the space, and the filesystem preallocates the necessary blocks and data structures, but there's no data on it until the application really needs to write the data in the future.
  - what P2P applications do but implemented much more efficiently by the filesystem
- several uses:
  - avoid applications doing it themselves inefficiently
  - improve fragmentation, since the blocks will be allocated at one time, as contiguously as possible.
  - ensure that applications have the space they will need, to avoid the filesystem getting full in the middle of an important operation.
- available via the libc posix_fallocate() interface.

Barriers on by default

- improves the integrity of the filesystem at the cost of some performance
- can be disabled with "mount -o barrier=0
- "The filesystem code must, before writing the [journaling] commit record, be absolutely sure that all of the transaction's information has made it to the journal.
- The kernel's block I/O subsystem makes this capability available through the use of barriers

管理Ext4文件系统

在 GNU/Linux 中，管理**Ext4**文件系统主要使用e2fsprogs中的一系列工具：

- 格式化为 Ext4 文件系统：mke2fs -T ext4 或 mkfs.ext4
- 升级 ext2/ext3 至 ext4
  – 确定要升级的 Ext3/Ext2 文件系统在unmount状态
  – 开启文件系统上 ext4 的新功能：tune2fs -O extents,uninit_bg,dir_index
  – fsck -pf
  – mount -t ext4
- 使用外部日志 (external journal)：可以提高文件系统效率
- 检查文件系统：e2fsck 或 fsck.ext4
- 设置常规文件系统检查
- 设置文件系统的系统管理员保留空间：Ext4 文件系统缺省保留 5% 的空间给系统管理员，该空间大小可以使用命令 tune2fs -m 或 tune2fs -r 改变
- 设置错误处理方案：tune2fs -e
- 显示文件系统卷标：e2label
- 改变文件系统卷标：e2label 或 tune2fs -L
- 设置文件系统 UUID：tune2fs -U

课后练习

- 了解文件系统的相关操作和实现

课程大作业

考查课：无笔试

内容： 截止时间：6月10日

1) 平时成绩 2) 课程报告

成绩分布

- 平时成绩（60%）：独立完成；提出并解决问题
  - 实验课: 课堂表现+课后实验报告（一周之内及时提交）
  - 课后作业：及时提交（一周之内），有自己的思考
  - 课程社区参与度（加分）

- 期末课程报告（40%）：
  - 相关材料（文档、视频等）提交（20%）：6月10日之前提交，有自己的见解
  - 课堂报告（20%）：6月10日中午12之前提交演示材料，要求思路清晰，课堂表达清楚

课程大作业——主题

- 主题：Linux内核相关
  - 课件内容，但更深入
  - 课件外主题，相关示例：
    - 内存回收分析
    - 网络子系统
    - Ext4数据结构及原理分析
    - RCU相关内核源码分析
    - 块设备驱动分析
    - …….

课程大作业——形式

- 作业形式：
  - 分析文档：鼓励自己画图表述思路
    - 原理分析
    - 数据结构
    - 源代码分析
      - 概述
      - 代码行注释
  - 实验报告：分析结果验证、问题及解决
    - 有特征的截图
    - 实验环境、设置
    - 实验原理、步骤、总结
    - 每个图都要有说明
  - 当堂报告：
    - Slides
    - 实验视频

课程大作业——组织

- 组织形式
  - 个人或小组
  - 小组注意事项：
    - 提交组内互评
    - 报告时按照人数计算时间
    - 组长加分（组织内容分配，组织文档撰写等）
    - **现场提问，注意基本知识的掌握**

课程大作业——建议和要求

- 要求
  - 标明内核版本5.0以上！
  - **不可大量原文、原码拷贝**——扣分！尤其是老版内容！
  - 建议：
    - 可以在网络上找相关资料，阅读后，对相关内容进行验证（网络资料有可能错误），然后对比新版修改相关部分也可以就新旧版本对照进行分析
  - 要有自己的心得、总结等
    - 文字描述以及图表：比如关系图、原理图等