# 内核源码阅读：文件系统

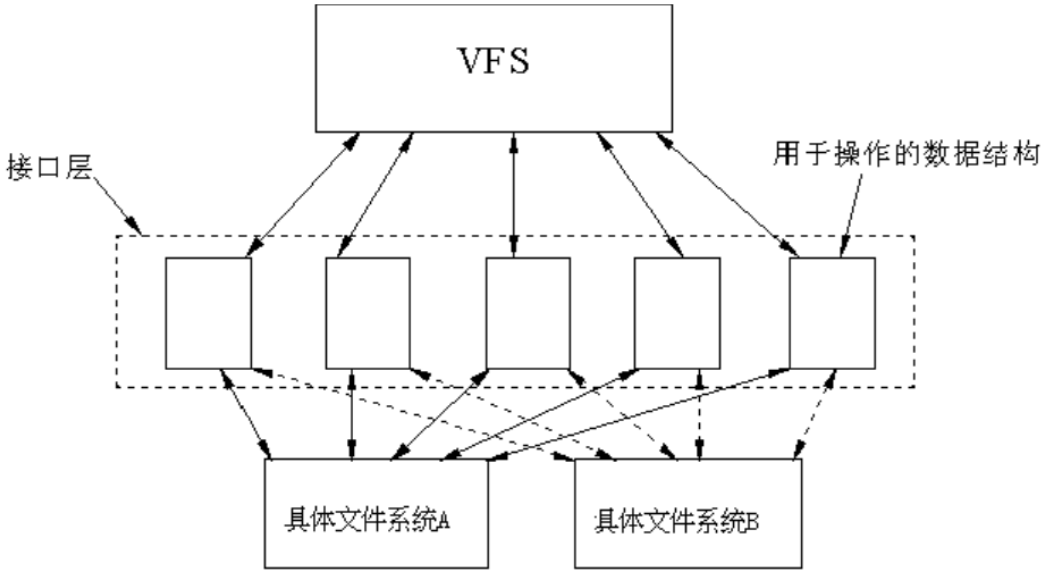
**江学强，PB16120100**

## 1. 虚拟文件系统（VFS）

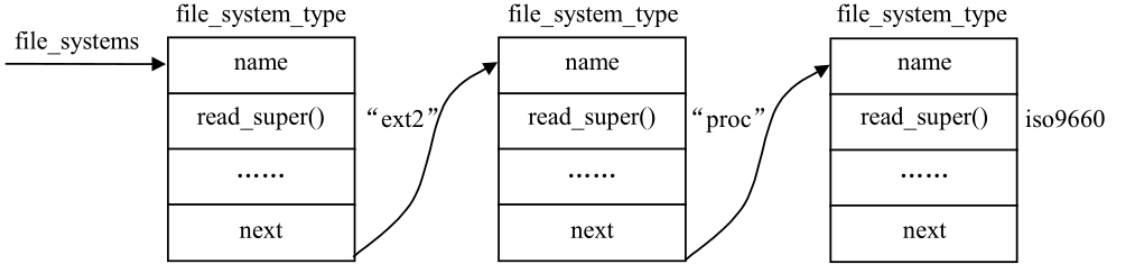
**VFS 的功能：**

VFS 是真实的文件系统和应用程序之间的抽象层，为应用程序提供统一的接口（file_operation 数据结构），这样应用程序就可以不用管具体的文件系统。在每次系统初始化期间，Linux 都首先要在 内存当中构造一棵 VFS 的目录树(在 Linux 的源代码里称之为 namespace)，以便于在内存中建立相应的数据结构。

VFS 核心功能：（1）抽象具体的文件系统，提供一种具体的数据结构来管理（具体后面叙述）；（2）接受统一的用户层的系统调用；（3）支持不同的文件系统间的访问；（3）提供文件系统和内核中其他模块的交互。




具体的文件系统在要在 VFS 中“注册”，其实就是每个实际文件系统对应了一个 file_system_type，这些 file_system_type 形成一个链表，链表头由 file_systems 指定，file_system_type 中包含文件系统的名称和指向对应 VFS 超级块读取例程的地址。




**VFS 重要的数据结构：**

VFS 中也有 super_block、inode、dentry 和 file 等标准文件系统模型中的数据结构，先叙述 VFS 中重要的数据结构和一些重要功能的实现：

*VFS 的 super_block*: 位于/include/linux/fs.h 中，VFS 的 super_block 用来存放系统中已安装的文件系统的有关信息

```
struct super_block {
    struct list_head    s_list;     /*list_head中包含了两个指针，s_list用来维护超级块链表    */
    dev_t           s_dev;      /* search index; _not_ kdev_t */
    unsigned char       s_dirt;
    unsigned char       s_blocksize_bits;
    unsigned long       s_blocksize;
    loff_t          s_maxbytes; /* Max file size */
    struct file_system_type *s_type;
    const struct super_operations   *s_op;
    const struct dquot_operations   *dq_op;
    const struct quotactl_ops   *s_qcop;
    const struct export_operations *s_export_op;
    unsigned long       s_flags;
    unsigned long       s_magic;
    struct dentry       *s_root;
    struct rw_semaphore s_umount;
    struct mutex        s_lock;
    int         s_count;
    atomic_t        s_active;
#ifdef CONFIG_SECURITY
    void                    *s_security;
#endif
    const struct xattr_handler **s_xattr;
```

```
    struct list_head    s_inodes;   /* all inodes */
    struct hlist_bl_head    s_anon;     /* anonymous dentries for (nfs) exporting */
#ifdef CONFIG_SMP
    struct list_head __percpu *s_files;
#else
    struct list_head    s_files;
#endif
    /* s_dentry_lru, s_nr_dentry_unused protected by dcache.c lru locks */
    struct list_head    s_dentry_lru;   /* unused dentry lru */
    int         s_nr_dentry_unused; /* # of dentry on lru */
    struct block_device *s_bdev;
    struct backing_dev_info *s_bdi;
    struct mtd_info     *s_mtd;
    struct list_head    s_instances;
    struct quota_info   s_dquot;    /* Diskquota specific options */
    int         s_frozen;
    wait_queue_head_t   s_wait_unfrozen;
    char s_id[32];              /* Informational name */
    u8 s_uuid[16];              /* UUID */
    void            *s_fs_info; /* Filesystem private info */
    fmode_t         s_mode;
    u32        s_time_gran;
    struct mutex s_vfs_rename_mutex;    /* Kludge */
    char *s_subtype;
```

包含的重要对象的解释：

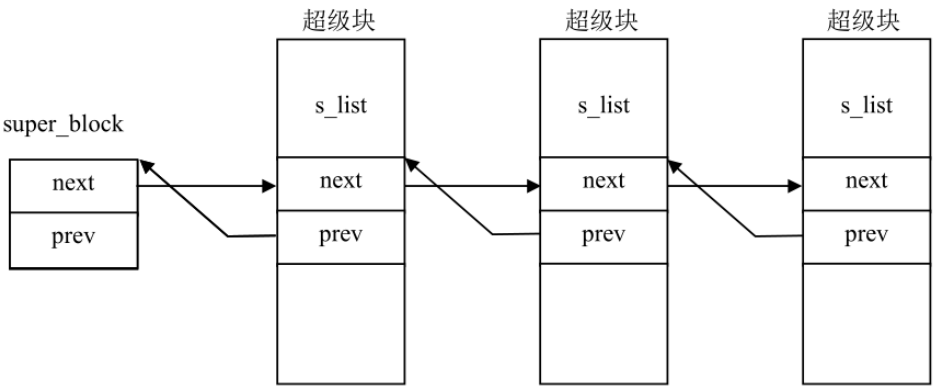
s_list：是一个包含两个指针的结构体，用来维护超级块的链表；

s_dev:设备标识符；s_dirt：脏位，标志超级块是否被修改；

s_blocksize_bits:块大小占用的位数；s_blocksize:总的块的大小；

s_maxbytes:文件最大字节数；s_type:文件系统的类型，是一个指向 file_system_type 的指针；s_op:指向特定文件系统的用于超级块操作的函数集合；dq_op：用于磁盘限额操作的函数集合；sqc_op：限额控制方法；s_flags：安装标志；s_lock：锁标志位，若置有效则其他进程不能修改这个超级块；s_magic：具体文件系统区分于其他文件系统的标志；s_root：具体文件系统的安装点。

每个安装（挂载）后的文件系统都有一个 super_block,所有的 super_block 以双向链表形式组织，如图所示：




super_block 中最重要的域是 s_op，指向超级块的函数表 super_operations，看到 super_operations 中定义了一堆函数指针，VFS 在 super_operations 中定义了统一的接口，具体的实现由具体的文件系统去实现。超级块操作函数执行文件系统和索引节点的底层操作，当文件系统需要对其超级块执行操作时，需要在超级块对象中寻找需要的操作方法。

*VFS 的 inode*：定义在/include/linux/fs.h 中

```
struct inode {
    /* RCU path lookup touches following: */
    umode_t         i_mode;
    uid_t           i_uid;
    gid_t           i_gid;
    const struct inode_operations   *i_op;
    struct super_block  *i_sb;
    spinlock_t      i_lock; /* i_blocks, i_bytes, maybe i_size */
    unsigned int        i_flags;
    struct mutex        i_mutex;
    unsigned long       i_state;
    unsigned long       dirtied_when;   /* jiffies of first dirtying */
    struct hlist_node   i_hash;
    struct list_head    i_wb_list;  /* backing dev IO list */
    struct list_head    i_lru;      /* inode LRU list */
    struct list_head    i_sb_list;
    union {
        struct list_head    i_dentry;
        struct rcu_head     i_rcu;
    };
    unsigned long       i_ino;
    atomic_t        i_count;
    unsigned int        i_nlink;
```

```
    unsigned int        i_nlink;
    dev_t           i_rdev;
    unsigned int        i_blkbits;
    u64         i_version;
    loff_t          i_size;
#ifdef __NEED_I_SIZE_ORDERED
    seqcount_t      i_size_seqcount;
#endif
    struct timespec     i_atime;
    struct timespec     i_mtime;
    struct timespec     i_ctime;
    blkcnt_t        i_blocks;
    unsigned short          i_bytes;
    struct rw_semaphore i_alloc_sem;
    const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
    struct file_lock    *i_flock;
    struct address_space    *i_mapping;
    struct address_space    i_data;
#ifdef CONFIG_QUOTA
    struct dquot        *i_dquot[MAXQUOTAS];
#endif
    struct list_head    i_devices;
```

```
    union {
        struct pipe_inode_info  *i_pipe;
        struct block_device *i_bdev;
        struct cdev     *i_cdev;
    };

    __u32           i_generation;

#ifdef CONFIG_FSNOTIFY
    __u32           i_fsnotify_mask; /* all events this inode cares about */
    struct hlist_head   i_fsnotify_marks;
#endif

#ifdef CONFIG_IMA
    atomic_t        i_readcount; /* struct files open RO */
#endif
    atomic_t        i_writecount;
#ifdef CONFIG_SECURITY
    void            *i_security;
#endif
#ifdef CONFIG_FS_POSIX_ACL
    struct posix_acl    *i_acl;
    struct posix_acl    *i_default_acl;
```

下面详细叙述 inode 中重要的一些域：inode 中有索引节点号 i_ino，在同一个文件系统中索引节点号是唯一的；i_uid，i_gid 标志了文件的所有者和文件所有者所在的组；i_mode 是文件访问权限的控制；inode 中与时间相关的域 i_atime、i_mtime 和 i_ctime 分别表示文件最后访问时间、最后修改时间和最后改变时间；i_blocks 指文件的块数，i_bytes 是文件使用的字节数；i_flags 是文件系统的标志；i_op 指向索引节点的操作表；i_state 表示 VFS 动态索引节点的状态，如果其值为 I_DIRTY 则表示该索引节点是“脏”的，因而对应的磁盘索引节点必须被更新。

与 super_block 类似，inode 中最重要的域是 i_op，指向索引节点的操作表，也是定义了一堆函数指针，提供了统一的操作的接口。

要注意 VFS 的索引节点是动态索引，与此相比，而具体文件系统的索引节点是静态索引，是放在磁盘上的，使用之前必须要先调到内存中。VFS 的索引节点会复制磁盘索引节点中包含的一些数据，例如文件占有的磁盘块数。

*VFS 中 dentry（目录项）*:位于 include/linux/dcache.h 中

每个文件除了有一个索引节点 inode 之外还有一个目录项 dentry，但是需要注意 dentry 是逻辑上的概念，磁盘上并没有对应的结构，所以一个索引节点可以对应好几个目录项 dentry。一个有效的 dentry 结构必定有一个 inode 结构，这是因为一个目录项要么代表着一个文件，要么代表着一个目录，而目录实际上也是文件。意思是只要 dentry 有效，则 dentry 中的 d_inode 指针一定指向一个 inode（考虑硬链接）。、

dentry 中一些重要的域：

d_count：引用计数器；d_parent：父目录的目录项；d_alias：索引节点别名的链表；d_subdirs：该目录项的子目录形成的链表；

```
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 */
    unsigned int d_count;       /* protected by d_lock */
    spinlock_t d_lock;      /* per dentry lock */
    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 */
    union {
        struct list_head d_child;   /* child of parent list */
        struct rcu_head d_rcu;
    } d_u;
    struct list_head d_subdirs; /* our children */
    struct list_head d_alias;   /* inode alias list */
};
```

同样的在同一个文件下还定义了 dentry_operations 数据结构，定义了 VFS 操作目录项的接口。

*文件对象（file）结构*：位于 include/linux/fs.h 中

file 结构是 VFS 中最后一个比较重要的结构，文件对象表示进程已经打开的一个文件。先分析数据结构中的域：

```
struct file {
    union {
        struct list_head    fu_list;
        struct rcu_head     fu_rcuhead;
    } f_u;
    struct path     f_path;
#define f_dentry    f_path.dentry
#define f_vfsmnt    f_path.mnt
    const struct file_operations    *f_op;
    spinlock_t      f_lock;  /* f_ep_links, f_flags, no IRQ */
#ifdef CONFIG_SMP
    int         f_sb_list_cpu;
#endif
    atomic_long_t       f_count;
    unsigned int        f_flags;
    fmode_t         f_mode;
    loff_t          f_pos;
    struct fown_struct  f_owner;
    const struct cred   *f_cred;
    struct file_ra_state    f_ra;

    u64         f_version;
```

```
#ifdef CONFIG_SECURITY
    void            *f_security;
#endif
    void            *private_data;
#ifdef CONFIG_EPOLL
    struct list_head    f_ep_links;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space    *f_mapping;
#ifdef CONFIG_DEBUG_WRITECOUNT
    unsigned long f_mnt_write_state;
#endif
};
```

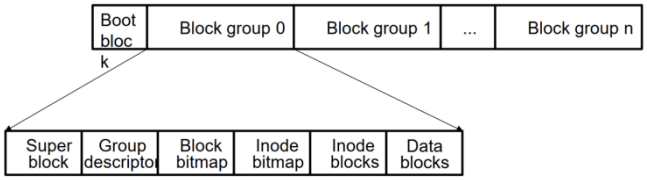
可以看出 file 的定义里面没有 f_dirty 这一项，所以 file 也是逻辑上的，磁盘上并没有对应这样的结构，文件通过 f_dentry 指针指向对应的目录项，目录项会指向对应的 inode，inode 中的 i_dirty 会标志文件是否被修改；

f_u 联合体维护了两个链表，一个是所有打开的 file 的链表，另一个是 file 打开后 free 之后的 file 链表；f_dentry 指向相关目录项的指针；f_vfsmnt 指向 VFS 挂载点的指针；f_mode 是文件的打开模式；f_pos 是文件的当前位置；f_count 是当前使用这个文件的进程数；

看完 file 的结构我觉得很奇怪因为看到 file 数据结构中的定义没有文件描述符这一项，在查了源码解读的书之后发现在 include/linux/sched.h 中定义了 files_struct 结构体，进程利用 files_struct 记录文件描述符的使用，每个进程有一个 files_struct，这是私有数据，files_struct 中的 fd 域指向文件对象的指针数组，数组的索引就是相应的文件描述符（file descriptor）。

## 2. Ext2 文件系统

Ext2 文件系统的磁盘布局：




Ext2 文件系统的数据结构：

ext2_super_block:定义在 incline/linux/ex2_fs.h 中

可以看出 ext2 文件系统的 super_block 记录了与这个具体文件系统相关的很多信息，s_inodes_count 记录文件系统中索引节点的总数，s_blocks_count 记录文件系统中数据块的总数等。已经看了 VFS 的 super_block 和 ext2_super_block，二者的关系在前面讲 VFS 的 super_block 时略微提了一下，具体文件系统的 super_block 必须要被读入内存才能使用，读入内存后的主要操作是把具体文件系统的 super_block 的一些数据填入 VFS 的 super_block 中。

```
struct ext2_super_block {
    __le32  s_inodes_count;     /* Inodes count */
    __le32  s_blocks_count;     /* Blocks count */
    __le32  s_r_blocks_count;   /* Reserved blocks count */
    __le32  s_free_blocks_count;    /* Free blocks count */
    __le32  s_free_inodes_count;    /* Free inodes count */
    __le32  s_first_data_block; /* First Data Block */
    __le32  s_log_block_size;   /* Block size */
    __le32  s_log_frag_size;    /* Fragment size */
    __le32  s_blocks_per_group; /* # Blocks per group */
    __le32  s_frags_per_group;  /* # Fragments per group */
    __le32  s_inodes_per_group; /* # Inodes per group */
    __le32  s_mtime;        /* Mount time */
    __le32  s_wtime;        /* Write time */
    __le16  s_mnt_count;        /* Mount count */
    __le16  s_max_mnt_count;    /* Maximal mount count */
    __le16  s_magic;        /* Magic signature */
    __le16  s_state;        /* File system state */
    __le16  s_errors;       /* Behaviour when detecting errors */
    __le16  s_minor_rev_level;  /* minor revision level */
    __le32  s_lastcheck;        /* time of last check */
    __le32  s_checkinterval;    /* max. time between checks */
    __le32  s_creator_os;       /* OS */
    __le32  s_rev_level;        /* Revision level */
```

Ext2_inode：定义在 include/linux/ext2_fs.h 中

```
struct ext2_inode {
    __le16  i_mode;     /* File mode */
    __le16  i_uid;      /* Low 16 bits of Owner Uid */
    __le32  i_size;     /* Size in bytes */
    __le32  i_atime;    /* Access time */
    __le32  i_ctime;    /* Creation time */
    __le32  i_mtime;    /* Modification time */
    __le32  i_dtime;    /* Deletion Time */
    __le16  i_gid;      /* Low 16 bits of Group Id */
    __le16  i_links_count;  /* Links count */
    __le32  i_blocks;   /* Blocks count */
    __le32  i_flags;    /* File flags */
    union {...
    } osd1;             /* OS dependent 1 */
    __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
    __le32  i_generation;   /* File version (for NFS) */
    __le32  i_file_acl; /* File ACL */
    __le32  i_dir_acl;  /* Directory ACL */
    __le32  i_faddr;    /* Fragment address */
    union {...
    } osd2;             /* OS dependent 2 */
};
```

可以看出具体文件系统结构中内核代码给出的注释很详细，不必详述具体数据的作用，但值得指出 Ext2 文件系统在 ext2_inode 中定义了指向数据块的指针数组 i_block，前 12 个是直接块指针，后面三个指针分别是一级指针、二级指针和三级指针。

组描述符（ext2_group_desc）:定义在 include/linux/ex2_fs.h 中

```
struct ext2_group_desc
{
    __le32  bg_block_bitmap;        /* Blocks bitmap block */
    __le32  bg_inode_bitmap;        /* Inodes bitmap block */
    __le32  bg_inode_table;     /* Inodes table block */
    __le16  bg_free_blocks_count;   /* Free blocks count */
    __le16  bg_free_inodes_count;   /* Free inodes count */
    __le16  bg_used_dirs_count; /* Directories count */
    __le16  bg_pad;
    __le32  bg_reserved[3];
};
```

在 ex2 磁盘布局中可以看到，紧挨着 super_block 的是组描述符表，每个组描述符用来描述一个块组的一些信息。

**3. Page cache 的预读和替换机制**

在 linux 系统中，为了加快文件的读写，内核中提供了 page cache 作为缓存。Page cache 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，page cache 规划就是用来缓存来自在文件系统下的任一文件的业务数据所存放的那些扇区。从操作系统的角度来看，Page Cache 可以看做是内存管理系统与文件系统之间的联系纽带。因此，Page Cache 管理是操作系统的一个重要组成部分，它的性能直接影响着文件系统和内存管理系统的性能。

其基本的实现原理与一般的 cache 相同，利用文件读写的时间和空间局部性，采用合适的预读和替换策略实现加速的功能

Page cache 的预读：对于每个文件的第一个读请求，系统读入所请求的页面并读入紧随其后的少数几个页面(不少于一个页面，通常是三个页面)，这时的预读称为同步预读。对于第二次读请求，如果所读页面不在 Cache 中，即不在前次预读的 group 中，则表明文件访问不是顺序访问，系统继续采用同步预读；如果所读页面在 Cache 中，则表明前次预读命中，操作系统把预读 group 扩大一倍，并让底层文件系统读入 group 中剩下尚不在 Cache 中的文件数据块，这时的预读称为异步预读。无论第二次读请求是否命中，系统都要更新当前预读 group 的大小。

Page cache 的替换：刚刚分配的 Cache 项链入到 inactive_list 头部，并将其状态设置为 active，当内存不够需要回收 Cache 时，系统首先从尾部开始反向扫描 active_list 并将状态不是 referenced 的项链入到 inactive_list 的头部，然后系统反向扫描 inactive_list，如果所扫描的项的处于合适的状态就回收该项，直到回收了足够数目的 Cache 项。

**4. Ext2 文件系统的数据块分配与预留机制**

当内核要分配一个新的数据块来保存 Ext2 普通文件的数据时就调用 ext2_get_block( )函数，这个函数定义在 fs/ext2/inode.c 中，ext2_get_block 函数通

过数据块寻址寻找空闲块并且在必要时调用 ext2_alloc_block()函数在 ext2 分区中实际搜索一个空闲的块。为了减少文件的碎片，Ext2 文件系统尽力在已分配给文件的最后一个块附近找一个新块分配给该文件。如果失败，Ext2 文件系统又在包含这个文件索引节点的块组中搜寻一个新的块。作为最后一个办法，可以从其他一个块组中获得空闲块。Ext2 文件系统使用数据块的预分配策略。文件并不仅仅获得所需要的块，而是获得一组多达 8 个邻接的块。ext2_inode_info 结构的 i_prealloc_count 域存放预分配给某一文件但还没有使用的数据块数，而 i_prealloc_block 域存放下一次要使用的预分配块的逻辑块号。当下列情况发生时，即文件被关闭时，文件被删除时，或关于引发块预分配的写操作而言，有一个写操作不是顺序的时候，就释放预分配但一直没有使用的块。具体的 ext2_get_block 函数太长，只分析一下这个函数参数的作用。

```
static int ext2_get_blocks(struct inode *inode,
                           sector_t iblock, unsigned long maxblocks,
                           struct buffer_head *bh_result,
                           int create)
```

参数 inode 指向文件的 inode 结构，iblock 表示文件中的逻辑块号，bh_result 是指向缓冲区首部的指针，参数 create 表示是否需要创建。

函数根据 iblock 算出数据块落在哪个索引区间，返回值为 0 表示出错，出错的原因可能是块号太大超过索引范围，之后调用 ext2_get_branch 函数完成文件内块号到设备块号的映射如果顺利完成则返回 NULL，如果在某一索引级发现索引表内的相应表项为 0，则说明这个数据块原来并不存在，此时，返回指向 Indirect 结构的指针。设备上具体物理块的分配，以及文件内数据块与物理块之间映射的建立，都是调用 ext2_alloc_branch（）函数完成的从 ext2_alloc_branch（）返回以后，我们已经从设备上分配了所需的数据块，包括用于间接索引的中间数据块。

**参考文献格式:**

[1] Abraham Silberschatz. 操作系统概念. 高等教育出版社, 2007.3.

[2] Daniel P. Bovet and Marco Cesati Understanding the Linux Kernel 3rd

[3]陈莉君 深入分析 Linux 内核源代码