中国人民大学 信息学院
RENMIN UNIVERSITY OF CHINA SCHOOL OF INFORMATION

操作系统内核分析与安全

6. 文件系统与磁盘管理

授课教师：游伟 副教授

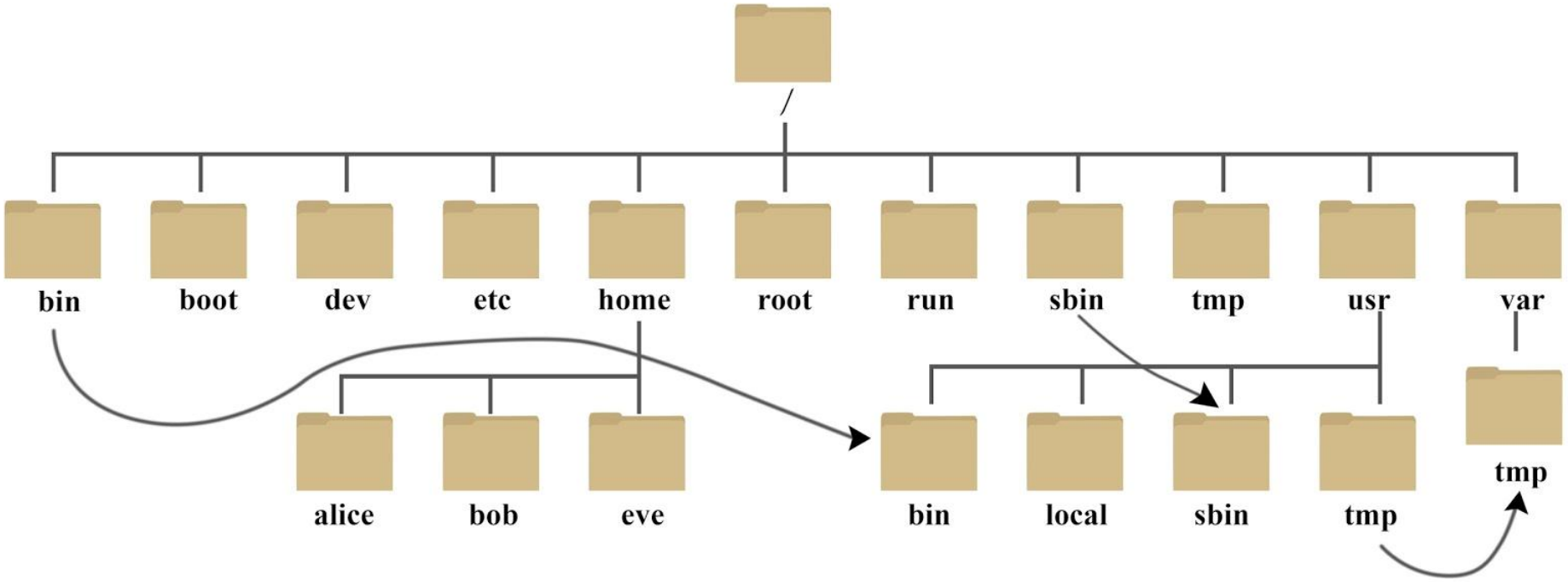
授课时间：周二14:00 – 15:30（立德楼807）

课程主页：https://www.youwei.site/course/kernel

本章内容

- Linux文件系统概述
- 虚拟文件系统VFS物理文件系统 ext4
- 文件open实现
- 实验一:添加一个文件系统
- 实验二:添加目录和文件到proc

Linux文件系统目录结构

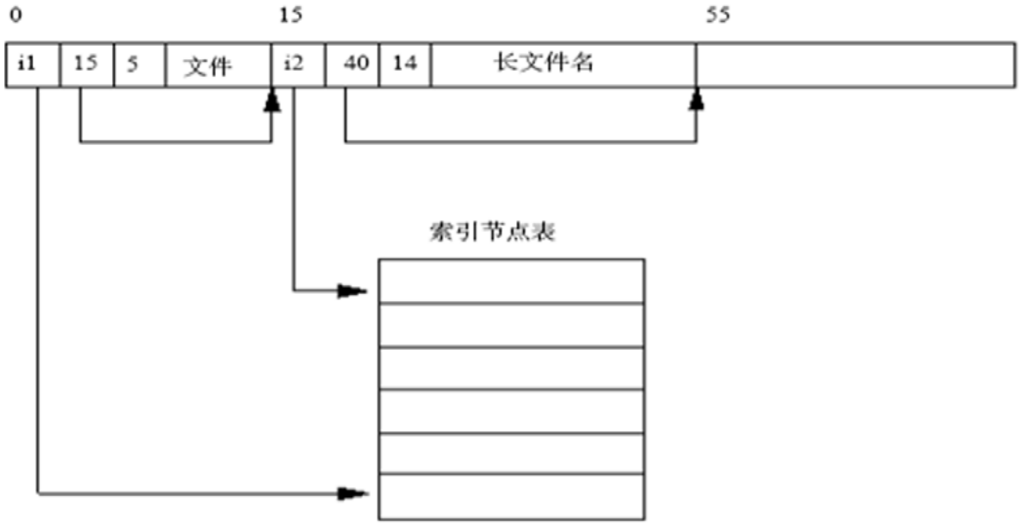


Linux文件系统概述

- Linux文件系统采用了多级目录的树型层次结构管理文件。树型结构的最上层是根目录，用/表示在根目录之下是各层目录和文件。
- Linux系统中的文件系统，不管是什么类型，都安装到一个目录下，并隐藏掉目录中原有的内容。这个目录叫做**安装目录**或者**安装点**。当文件系统卸载掉时，目录中的原有内容将再一次的显示出来。
- Linux通过使用一组通用的 **API函数**，可以在许多种存储设备上支持许多种文件系统。

Linux文件系统概述

- Linux缺省的文件系统(ext2、ext3、ext4)继承了UINX，把文件名和文件控制信息分开管理，文件控制信息单独组成一个称为索引节点(inode.个数据结构)。每个文件对应一个inode，它们有唯一的编号，称为inode号(一个整型值)。
- 目录项主要由文件名和inode号组成。



文件的类型

- 普通文件
  - 文件名最长不能超过255个字符
  - 可以用除保留字符以外的任何字符给文件命名
  - 强烈建议不要使用非打印字符、空白字符(空格和制表符)和shell 命令保留字符
  - 扩展名对LINUX系统来说没有任何意义
  - 可以任意给文件名加上你自己或应用程序定义的扩展名(e.g..c file extension is required byC compilers)
- 目录文件
  - 是文件系统中一个目录所包含的目录项组成的文件。目录文件只允许系统进行修改。用户进程可以读取目录文件，但不能对它们进行修改。两个特殊的目录项“.”代表目录本身，”..”表示父目录。

文件的类型

- 字符设备文件和块设备文件。Linux把对设备的I/O作为对文件的读取/写入操作内核提供了对设备处理和对文件处理的统一接口。
  - fd0 (for floppy drive 0)
  - hda (for harddisk a)
  - lp0 (for line printer 0)
  - tty (for teletype terminal)
- 管道(FIFO)文件:用于在进程间传递数据。Linux对管道的操作与文件操作相同，它把管道做为文件进行处理。
- 链接文件:又称符号链接文件，它提供了共享文件的一种方法。
- socket文件

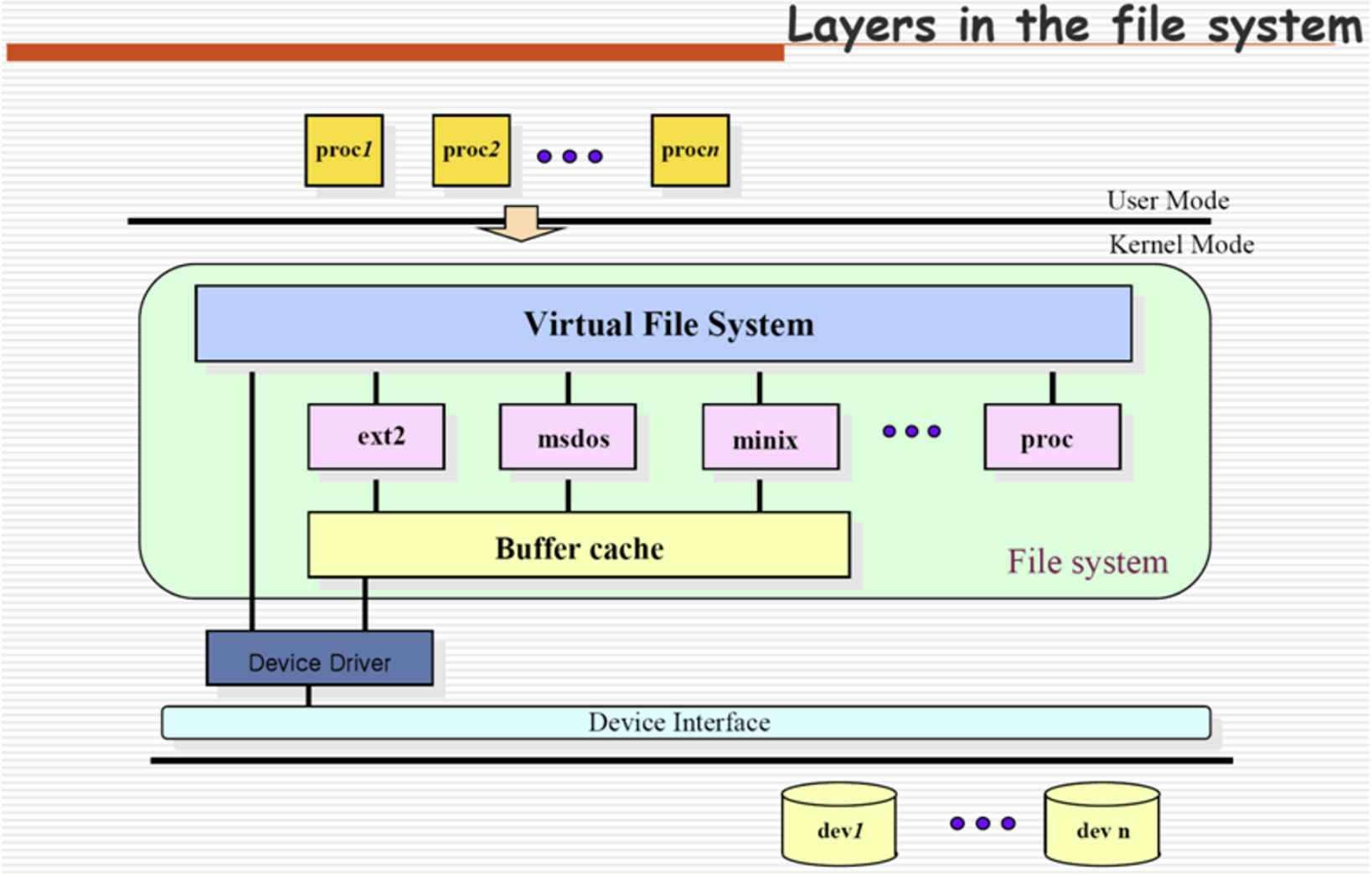
文件系统的类型

- 文件系统分三大类:
  - 基于磁盘的文件系统，如ext2/ext3/ext4、VFAT、NTFS等。
  - 网络文件系统，如NFS等。
  - 特殊文件系统，如pr0c文件系统、devfs、sysfs(/sys)等。
- 支持多种不同类型的文件系统是Linux操作系统的一大特色。
- Linux在标准内核中已支持的文件系统超过50种，
- Linux的标准文件系统是ext2或ext3或ext4，系统把它的磁盘分区做为系统的根文件系统。

VFS虚拟文件系统

- Linux把各种不同的物理文件系统的所有特性进行抽象，建立起一个面向各种物理文件系统的转换机制，通过这个转换机制，把各种不同物理文件系统转换为一个具有统一共性的虚拟文件系统这种转换机制称为**虚拟文件系统转换VFS(Virtual Filesystem Switch/System)**。
- VFS并不是一种实际的文件系统。ext2/ext4等物理文件系统是存在于外存空间的，而**VFS仅存在于内存**。
- 在 VFS上面，是对诸如 open、close、read 和 write 之类的函数的一个通用 AP|抽象。在 VFS 下面是文件系统抽象，它定义了上层函数的实现方式。文件系统的源代码可以在 linux/fs 中找到。

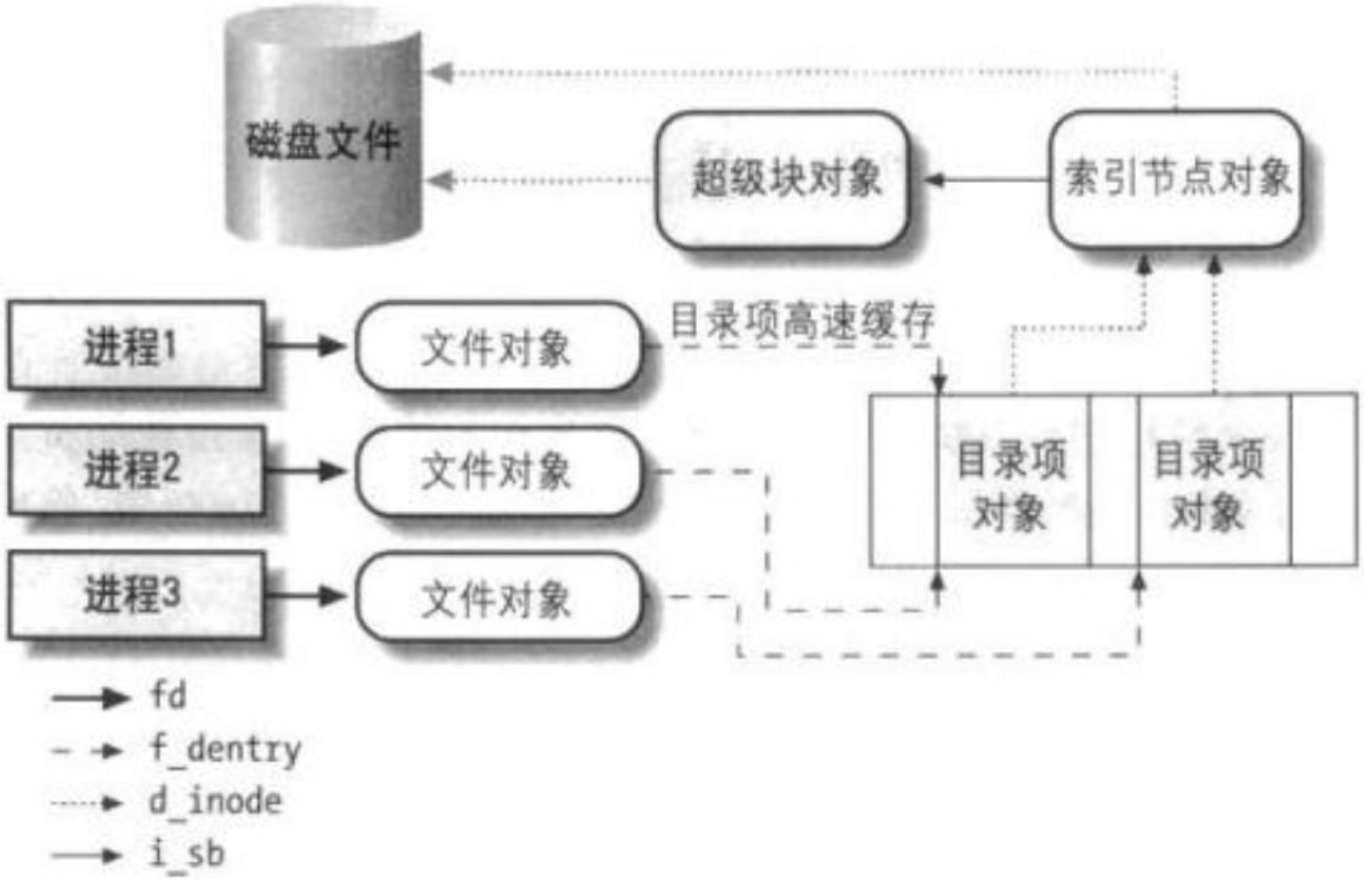
VFS虚拟文件系统


Layers in the file system
proc1
proc2
procn
User Mode
Kernel Mode
Virtual File System
ext2
msdos
minix
proc
Buffer cache
File system
Device Driver
Device Interface
dev1
dev n

VFS文件系统的结构

- VFS根据不同的文件系统抽象出了一个通用的文件模型。通用的文件模型由四种数据对象组成:
  - 超级块对象 superblock:存储已安装文件系统的信息，通常对应磁盘文件系统的文件系统超级块或控制块。
  - 索引节点对象 inode object:存储某个文件的信息!通常对应磁盘文件系统的文件控制块。
  - 目录项对象dentry object :dentry对象主要是描述一个目录项，是路径的组成部分。
  - 文件对象 file object:存储一个打开文件和一个进程的关联信息。只要文件一直打开，这个对象就一直存在与内存。

进程与VFS的交互


磁盘文件
超级块对象
索引节点对象
进程1
进程2
进程3
文件对象
文件对象
文件对象
目录项高速缓存
目录项对象
目录项对象
fd
f_dentry
d_inode
i_sb

VFS的超级块

- 超级块superblock是文件系统中描述整体组织和结构的信息体。
- VFS把不同文件系统中的整体组织和结构信息进行抽象后形成了兼顾不同文件系统的统一的超级块结构。
- VFS超级块是各种具体文件系统在安装时建立的，并在卸载时被自动删除。
- Linux中对于每种已安装的文件系统，在内存中都有与其对应的超级块。VFS超级块中的数据主要来自该文件系统的超级块。
- VFS超级块的数据结构定义是super_block结构。

super_block

```
struct super_block {
        struct list_head        s_list;         /* Keep this first */
        dev_t                   s_dev;          /* search index; _not_ kdev_t */
        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_iflags;       /* internal SB_I_* flags */
        unsigned long           s_magic;
        struct dentry           *s_root;
        struct rw_semaphore     s_umount;
        int                     s_count;
        atomic_t                s_active;
#ifdef CONFIG_SECURITY
        void                    *s_security;
#endif
        const struct xattr_handler * const *s_xattr;
#ifdef CONFIG_FS_ENCRYPTION
        const struct fscrypt_operations *s_cop;
        struct fscrypt_keyring  *s_master_keys; /* master crypto keys in use */
#endif
#ifdef CONFIG_FS_VERITY
        const struct fsverity_operations *s_vop;
#endif
#if IS_ENABLED(CONFIG_UNICODE)
        struct unicode_map *s_encoding;
        __u16 s_encoding_flags;
#endif
        struct hlist_bl_head    s_roots;        /* alternate root dentries for NFS */
        struct list_head        s_mounts;       /* list of mounts; _not_ for fs use */
        struct block_device     *s_bdev;        /* can go away once we use an accessor for @s_bdev_file */
        struct file             *s_bdev_file;
        struct backing_dev_info *s_bdi;
```

super_block

- s_list：将该super_block链接到super_blocks变量指向的链表中。
- s_blocksize：系统中文件的最小块大小。
- s_blocksize_bits：表示s_blocksize需要的位数。
- s_type：提供文件系统的mount、init_fs_context等回调函数。
- s_op：提供alloc_inode、destroy_inode等回调函数。
- s_root：文件系统的root文件。
- s_inodes：文件系统的inode组成的链表的头。
- s_instances：将它链接到同一种文件系统的组成的链表。
- s_mounts：挂载它的mount对象组成的链表的头。
- s_fs_info：私有数据。

super_block示例

字段名	作用说明	示例值/相关结构体
`s_list`	将 `super_block` 链接到全局 `super_blocks` 链表，用于内核管理所有已挂载的文件系统实例。	所有文件系统的 `super_block` 通过此字段形成全局链表。
`s_blocksize`	文件系统的逻辑块大小（字节单位），定义文件系统 I/O 的最小单位。	常见值为 `4096`（4KB），由底层文件系统初始化时设置。
`s_blocksize_bits`	`s_blocksize` 的二进制位数，用于位运算优化（例如块偏移计算）。	若 `s_blocksize=4096`，则 `s_blocksize_bits=12`（因 `2^12=4096`）。
`s_type`	指向文件系统的类型描述符 `file_system_type`，包含文件系统名称、挂载方法等通用信息。	所有文件系统（如 ext4、proc）均需注册自己的 `file_system_type`，VFS 通过此字段调用具体实现。
`s_op`	超级块操作函数表（`super_operations`），定义 VFS 操作文件系统的通用方法（如创建/销毁 inode）。	具体文件系统需实现这些回调函数（如 `alloc_inode`、`destroy_inode`）。
`s_root`	指向该文件系统根目录的 `dentry`（目录条目），是文件系统树形结构的起点。	VFS 通过 `dentry` 和 `inode` 解析路径（如 `/etc/passwd`）。
`s_inodes`	链入该超级块管理的所有 `inode` 的链表头。当文件被打开或关闭时，inode 会被添加或移出链表。	用于内核快速遍历该文件系统的所有活跃 inode。
`s_instances`	将该 `super_block` 链接到同类型文件系统的实例链表（由 `file_system_type->fs_supers` 管理）。	所有同类型文件系统的 `super_block` 通过此字段形成链表。
`s_mounts`	挂载点链表头，链接所有通过 `mount` 挂载到该文件系统的 `vfsmount` 结构。	支持文件系统嵌套挂载（如 bind mount）。
`s_fs_info`	指向具体文件系统的私有数据（如 ext4 的 `ext4_sb_info`），VFS 不关心其内容，仅提供存储指针。	由具体文件系统初始化和使用，VFS 仅负责传递指针。

VFS超级块的操作

- 在系统运行中，VFS要建立、撤消一些VFS inode，还要对VFS超级块进行一些必要的操作。这些操作由一系列操作函数实现。
- 不同类型的文件系统的组织和结构不同，完成同样功能的操作函数的代码不同，每种文件系统都有自己的操作函数。
- 如何在对某文件系统进行操作时就能调用该文件系统的操作函数呢?这是由VFS接口通过转换实现的。
- 在VFS超级块中s_op是一个指向super_operations结构的指针，super_operations中包含着一系列的操作函数指针，即这些操作函数的入口地址。
- 每种文件系统VFS超级块指向的super_operations中记载的是该文件系统的操作函数的入口地址。只需使用它们各自的超级块成员项s_op，以统一的函数调用形式:s_op->read_inode()就可以分别调用它们各自的读inode操作函数。

super_operations

```
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);
        void (*free_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 *, enum freeze_holder who);
        int (*freeze_fs) (struct super_block *);
        int (*thaw_super) (struct super_block *, enum freeze_holder who);
        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 *);

        int (*show_options)(struct seq_file *, struct dentry *);
        int (*show_devname)(struct seq_file *, struct dentry *);
        int (*show_path)(struct seq_file *, struct dentry *);
        int (*show_stats)(struct seq_file *, struct dentry *);
#ifdef CONFIG_QUOTA
        ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
        ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
        struct dquot __rcu **(*get_dquots)(struct inode *);
#endif
        long (*nr_cached_objects)(struct super_block *,
                                  struct shrink_control *);
        long (*free_cached_objects)(struct super_block *,
                                    struct shrink_control *);
        void (*shutdown)(struct super_block *sb);
};
```

super_operations

- alloc_inode: 在超级块中分配并初始化一个新的 inode 实例。
- destroy_inode: 销毁一个 inode 的内容，释放其所占资源但不一定释放内存。
- free_inode: 释放 inode 所占用的内存，将其彻底清除出系统。
- dirty_inode: 将 inode 标记为“脏”状态，表明其内容已被修改，需要写回磁盘，并传递相关标志。
- write_inode: 将 inode 中的修改数据写回到磁盘中，确保数据同步。
- drop_inode: 判断并处理 inode 的引用计数，决定是否可以丢弃并释放该 inode。
- evict_inode: 从 inode 缓存中移除并清理一个 inode，通常在其不再被引用时调用。
- put_super: 释放对超级块的引用，当超级块不再使用时执行清理操作。
- sync_fs: 将整个文件系统的所有脏数据同步写入磁盘，保证数据一致性。
- freeze_super: 冻结超级块，阻止对文件系统进行写入操作，通常用于系统快照或备份。
- freeze_fs: 冻结整个文件系统，暂时阻止所有写入操作以确保一致性。
- thaw_super: 解冻超级块，恢复对文件系统的写入操作，并解除冻结状态。
- unfreeze_fs: 解冻整个文件系统，使其恢复正常的读写作。

super_operations

- unfreeze_fs: 解冻整个文件系统，使其恢复正常的读写操作。
- statfs: 收集并返回文件系统的统计信息，如总容量、可用空间等。
- remount_fs: 重新挂载文件系统，允许修改挂载时的选项或参数。
- umount_begin: 在卸载文件系统前执行必要的预处理步骤，为卸载做好准备。
- show_options: 在序列文件中显示挂载选项，通常用于 /proc 文件系统中的展示。
- show_devname: 展示与文件系统关联的设备名称信息。
- show_path: 显示文件系统的路径信息，帮助识别其挂载点。
- show_stats: 输出文件系统的统计数据，提供性能和使用情况的信息。
- quota_read (仅在启用配额时): 读取并返回文件系统的配额数据。
- quota_write (仅在启用配额时): 将更新后的配额数据写入文件系统。
- get_dquots (仅在启用配额时): 获取与指定 inode 关联的配额对象列表。
- nr_cached_objects: 计算并返回当前缓存对象的数量，辅助资源管理与回收。
- free_cached_objects: 释放缓存中不再需要的对象，腾出内存资源。
- shutdown: 执行文件系统关闭前的清理操作，确保所有数据安全写回并释放资源。

VFS的inode对象

- Linux以ext4做为基本的文件系统，所以它的虚拟文件系统VFS中也设置了inode结构。
- 物理文件系统的inode在外存中并且是长期存在的， VFS的inode对象在内存中，它仅在需要时才建立，不再需要时撤消。
- 物理文件系统的inode是静态的，而VFS的inode是一种动态结构。
- inode结构定义在文件<include/linux/fs.h>中。

inode

```
struct inode {
        umode_t                 i_mode;
        unsigned short          i_opflags;
        kuid_t                  i_uid;
        kgid_t                  i_gid;
        unsigned int            i_flags;

#ifdef CONFIG_FS_POSIX_ACL
        struct posix_acl        *i_acl;
        struct posix_acl        *i_default_acl;
#endif

        const struct inode_operations   *i_op;
        struct super_block      *i_sb;
        struct address_space    *i_mapping;

#ifdef CONFIG_SECURITY
        void                    *i_security;
#endif

        /* Stat data, not accessed from path walking */
        unsigned long           i_ino;
        /*
         * Filesystems may only read i_nlink directly.  They shall use the
         * following functions for modification:
         *
         *    (set|clear|inc|drop)_nlink
         *    inode_(inc|dec)_link_count
         */
        union {
                const unsigned int i_nlink;
                unsigned int __i_nlink;
        };
        dev_t                   i_rdev;
        loff_t                  i_size;
        time64_t                i_atime_sec;
        time64_t                i_mtime_sec;
        time64_t                i_ctime_sec;
        u32                     i_atime_nsec;
```

inode

- i_mode：定义了文件的类型和访问权限，决定了该 inode 对应的对象是常规文件、目录、符号链接等，以及相应的读写执行权限。
- i_uid：表示文件的所有者（用户ID），用于权限检查和安全控制，确保只有合适的用户能够访问或修改文件。
- i_gid：表示文件所属的组（组ID），配合 i_uid 一同参与文件权限管理，支持多用户环境下的访问控制。
- i_ino：是 inode 的唯一标识号，在文件系统中用于唯一标识一个文件，即使文件名可能改变，该号码仍保持不变。
- i_size：存储文件的字节数，反映了文件的实际大小，在读写操作、空间分配等方面起到关键作用。
- i_sb：指向所属超级块（super_block）的指针，建立了 inode 与整个文件系统之间的关联，使得文件系统的全局信息能够被访问。
- i_op：指向 inode 操作函数集的指针，通过该接口调用具体文件系统实现的操作（如创建、删除、读写等），实现了 VFS 层与底层文件系统的解耦。
- i_mapping：指向地址空间（address_space）结构，用于管理文件内容在内存中的缓存映射，直接影响文件的 I/O 性能。
- i_count：内核对该 inode 的引用计数，管理该对象的生命周期，确保在多进程或多线程环境中 inode 不会被提前释放，同时也便于缓存管理。

inode

```
❯ stat /etc/passwd
  文件：/etc/passwd
  大小：3762            块：8          IO 块大小：4096   普通文件
设备：802h/2050d        Inode：74191564    硬链接：1
权限：(0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
访问时间：2025-03-24 21:30:01.723704623 +0800
修改时间：2024-10-26 21:05:43.563292314 +0800
变更时间：2024-10-26 21:05:43.595292484 +0800
创建时间：2024-10-26 21:05:43.563292314 +0800
```

VFS inode 字段	实例化值/描述	对应 `stat` 输出项
`i_mode`	`S_IFREG | 0644`（普通文件，权限 `rw-r--r--`） 二进制值：`0b1000 0001 1010 0100`（十六进制 `0x81A4`）	权限：(0644/-rw-r--r--)
`i_uid`	`0`（root 用户）	Uid: (0/root)
`i_gid`	`0`（root 组）	Gid: (0/root)
`i_ino`	`74191564`（该文件的唯一 inode 编号）	Inode：74191564
`i_size`	`3762`（文件大小，单位：字节）	大小：3762

inode操作

- VFS提供的inode操作函数在inode operations结构中，它们由一系列对inode进行操作的函数指针组成，inode结构体中iop指向inode_operations结构。

```
struct inode_operations {
        struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
        const char * (*get_link) (struct dentry *, struct inode *, struct delayed_call *);
        int (*permission) (struct mnt_idmap *, struct inode *, int);
        struct posix_acl * (*get_inode_acl)(struct inode *, int, bool);

        int (*readlink) (struct dentry *, char __user *,int);

        int (*create) (struct mnt_idmap *, struct inode *,struct dentry *,
                       umode_t, bool);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct mnt_idmap *, struct inode *,struct dentry *,
                        const char *);
        int (*mkdir) (struct mnt_idmap *, struct inode *,struct dentry *,
                      umode_t);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct mnt_idmap *, struct inode *,struct dentry *,
                      umode_t,dev_t);
        int (*rename) (struct mnt_idmap *, struct inode *, struct dentry *,
                       struct inode *, struct dentry *, unsigned int);
        int (*setattr) (struct mnt_idmap *, struct dentry *, struct iattr *);
        int (*getattr) (struct mnt_idmap *, const struct path *,
```

inode_operations

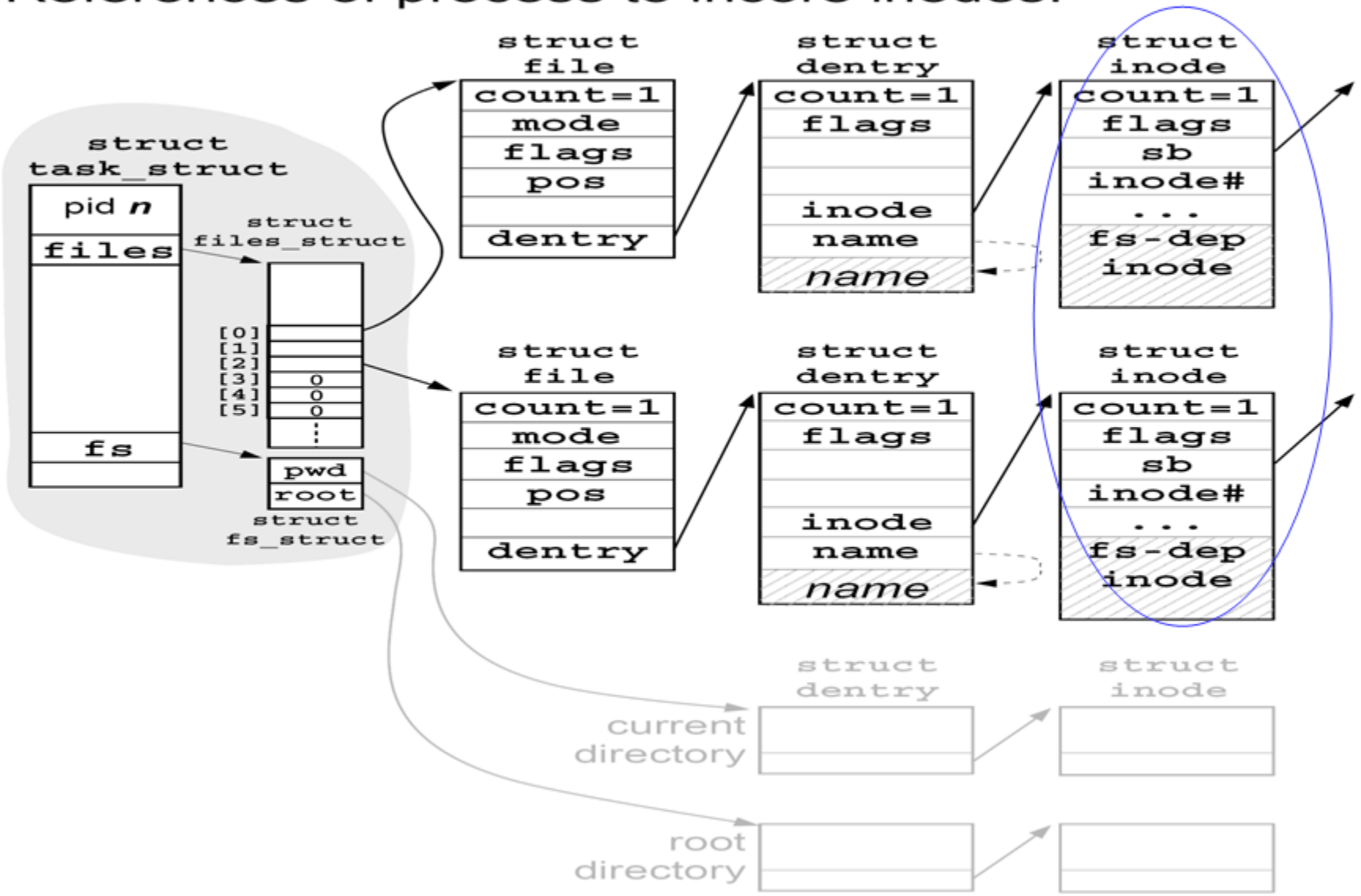
- lookup 用于在给定的 inode 中查找指定 dentry 并返回匹配结果；
- get_link 用于获取符号链接的目标路径；
- permission 用于检查对 inode 的访问权限；
- get_inode_acl 用于获取 inode 的访问控制列表；
- readlink 用于读取符号链接中的内容；
- create 用于在目录中创建新文件，并设置相应的权限；
- link 用于为已有的 inode 创建新的硬链接；
- unlink 用于删除指定的文件；
- symlink 用于在目录中创建符号链接；
- mkdir 用于创建新目录；

inode_operations

- rmdir 用于删除目录；
- mknod 用于在目录中创建特殊文件（如设备文件）；
- rename 用于重命名或移动文件和目录；
- setattr 用于设置文件或目录的属性；
- getattr 用于获取文件或目录的状态信息；
- listxattr 用于列出文件或目录的扩展属性；
- fiemap 用于获取文件的物理块映射信息；
- update_time 用于更新 inode 的时间戳；
- atomic_open 用于以原子方式打开文件以保证并发安全；
- tmpfile 用于创建临时文件；
- get_acl 用于获取目录项的访问控制列表；
- set_acl 用于设置目录项的访问控制列表；
- fileattr_set 用于设置文件的扩展属性；
- fileattr_get 用于获取文件的扩展属性；
- get_offset_ctx 用于获取与文件 I/O 操作相关的偏移上下文信息。

Processes and incore inodes

References of process to incore inodes:



目录项对象dentry object

- 每个文件除了有一个索引节点inode数据结构外还有一个目录项dentry数据结构。
- 每个dentry代表路径中的一个特定部分。如：/、bin都属于目录项对象。
- 目录项也可包括安装点，如:/mnt/cdrom/foo，1、mnt、cdrom、foo都属于目录项对象。
- 目录项对象作用是帮助实现文件的快速定位，还起到缓冲作用。
  - 缓冲：它在内核中缓存了文件名到 inode 的映射信息；因此，当系统需要再次访问相同文件时，可以直接从内存中获取对应的 inode，而不必每次都重新读取磁盘目录

目录项对象dentry object

```
struct dentry {
        /* RCU lookup touched fields */
        unsigned int d_flags;           /* protected by d_lock */
        seqcount_spinlock_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 */
        /* --- cacheline 1 boundary (64 bytes) was 32 bytes ago --- */

        /* Ref lookup also touches following */
        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 */
        /* --- cacheline 2 boundary (128 bytes) --- */
        struct lockref d_lockref;       /* per-dentry lock and refcount
                                         * keep separate from RCU lookup area if
                                         * possible!
                                         */

        union {
                struct list_head d_lru;         /* LRU list */
                wait_queue_head_t *d_wait;      /* in-lookup ones only */
        };
        struct hlist_node d_sib;        /* child of parent list */
        struct hlist_head d_children;   /* our children */
        /*
         * d_alias and d_rcu can share memory
         */
        union {
                struct hlist_node d_alias;      /* inode alias list */
                struct hlist_bl_node d_in_lookup_hash;  /* only for in-lookup ones */
                struct rcu_head d_rcu;
        } d_u;
};
```

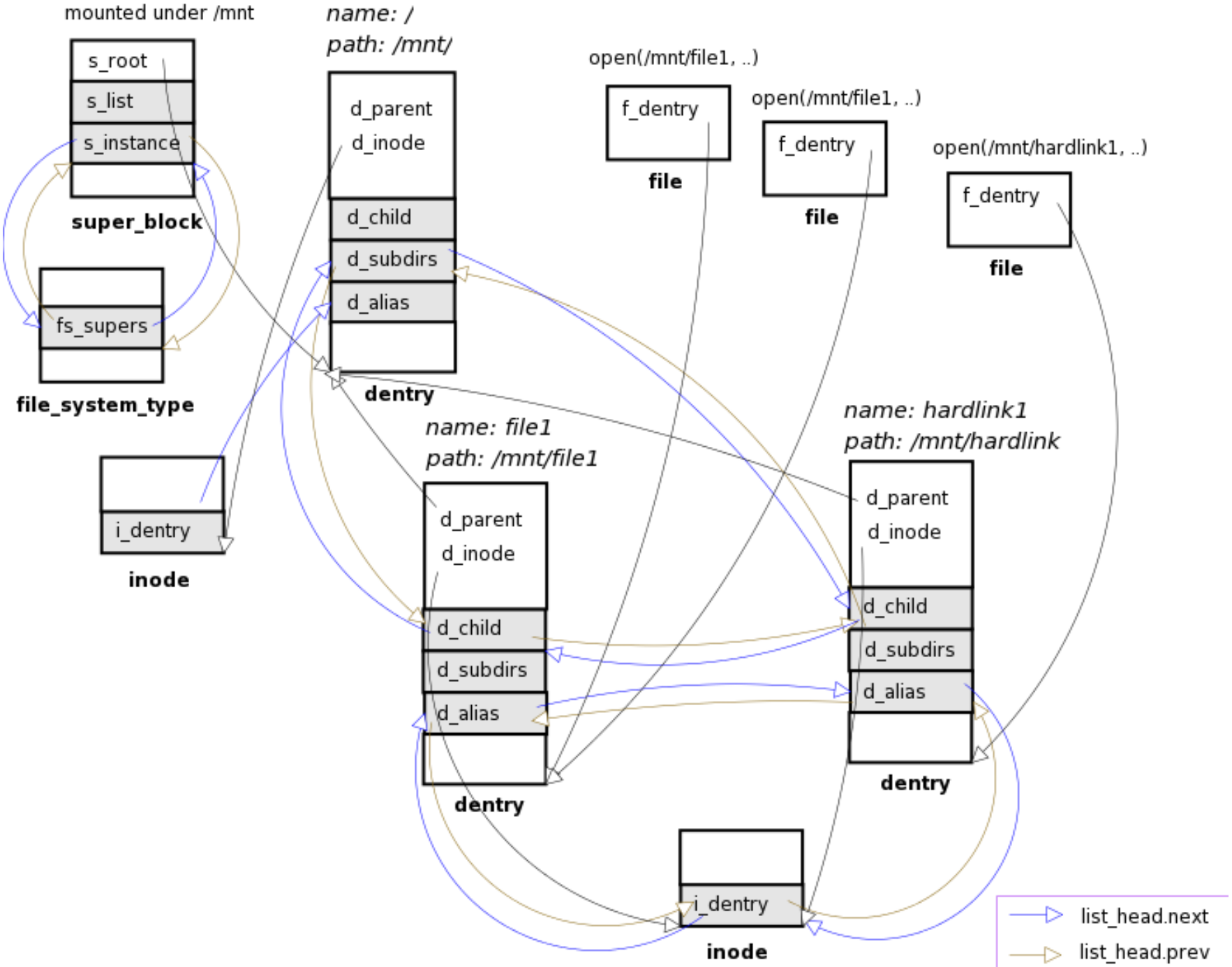


Fig: Relationships between the VFS objects

目录项对象dentry object

- d_name 表示目录项中存储的文件名；
- d_inode 指向该名称对应的 inode，标识文件的具体对象；
- d_parent 指向父目录的 dentry，用于构建文件系统的层次结构；
- d_hash 用于在哈希表中快速定位该目录项；
- d_children 保存该目录项下的所有子目录和文件的列表；
- d_flags：标识目录项的状态标志，反映其缓存状态和其他属性；
- d_op：指向与该目录项相关的操作函数集，用于实现特定的 dentry 功能；
- d_sb：指向该目录项所在文件系统的超级块，建立与文件系统全局信息的联系；
- d_lockref：提供目录项的锁定和引用计数机制，确保多线程环境下安全访问。

目录项对象dentry object

字段名	实例化值/行为描述	关联场景
`d_name`	文件名和哈希值，存储字符串 `"passwd"`，并通过哈希算法优化查找性能。	用户访问 `/etc/passwd` 时，内核通过 `d_name` 匹配文件名。
`d_inode`	指向该文件对应的 `inode`（即 `i_ino=74191564` 的 inode）。	通过 `d_inode` 访问文件的元数据（如权限、大小）和内容。
`d_parent`	指向父目录的 `dentry`（即 `/etc` 目录的目录项）。	路径解析时，从根目录 `/` 逐级查找 `etc`，再通过 `d_parent` 找到 `passwd` 的 dentry。
`d_hash`	将该 dentry 链接到内核哈希表（`dentry_hashtable`），以 `文件名 + 父目录 dentry` 为键值快速定位。	避免重复解析路径，例如多次访问 `/etc/passwd` 时直接命中缓存。
`d_children`	空。	`/etc/passwd` 是文件，无子项；若为目录（如 `/etc`），则 `d_children` 包含 `passwd` 等子项 dentry。
`d_sb`	指向所属文件系统的 `super_block`（即 ext4 的 `super_block`）。	通过 `d_sb` 访问文件系统的全局信息（如块大小、根目录 inode）。

目录项对象dentry object操作函数

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, unsigned int);
        int (*d_weak_revalidate)(struct dentry *, unsigned int);
        int (*d_hash)(const struct dentry *, struct qstr *);
        int (*d_compare)(const struct dentry *,
                         unsigned int, const char *, const struct qstr *);
        int (*d_delete)(const struct dentry *);
        int (*d_init)(struct dentry *);
        void (*d_release)(struct dentry *);
        void (*d_prune)(struct dentry *);
        void (*d_iput)(struct dentry *, struct inode *);
        char *(*d_dname)(struct dentry *, char *, int);
        struct vfsmount *(*d_automount)(struct path *);
        int (*d_manage)(const struct path *, bool);
        struct dentry *(*d_real)(struct dentry *, enum d_real_type type);
} ____cacheline_aligned;
```

目录项对象dentry object操作函数

- d_revalidate 用于检查并更新目录项是否仍然有效；
- d_weak_revalidate 则提供了一种较宽松的验证机制；
- d_hash 用于根据目录项内容计算或更新哈希值；
- d_compare 用于比较目录项以确认名称匹配；
- d_delete 用于判断目录项是否应该被删除；
- d_init 负责初始化新创建的目录项；
- d_release 在目录项释放时执行清理工作；
- d_prune 用于清除不再使用的目录项；
- d_iput 用于处理目录项与 inode 的关联释放；
- d_dname 用于生成目录项的显示名称；
- d_automount 在自动挂载时返回相应的挂载点；
- d_manage 用于管理目录项状态或调整目录结构；
- d_real 用于获取目录项的真实底层对象。

文件对象file

- 文件对象file表示进程已打开的文件，只有当文件被打开时才在内存中建立file对象的内容。
- 该对象由相应的open()系统调用创建，由close()系统调用销毁。

```
struct file {
        file_ref_t                      f_ref;
        spinlock_t                      f_lock;
        fmode_t                         f_mode;
        const struct file_operations    *f_op;
        struct address_space            *f_mapping;
        void                            *private_data;
        struct inode                    *f_inode;
        unsigned int                    f_flags;
        unsigned int                    f_iocb_flags;
        const struct cred               *f_cred;
        /* --- cacheline 1 boundary (64 bytes) --- */
        struct path                     f_path;
        union {
                /* regular files (with FMODE_ATOMIC_POS
                struct mutex            f_pos_lock;
                /* pipes */
                u64                     f_pipe;
        };
        loff_t                          f_pos;
#ifdef CONFIG_SECURITY
        void                            *f_security;
#endif
        /* --- cacheline 2 boundary (128 bytes) --- */
        struct fown_struct              *f_owner;
        errseq_t                        f_wb_err;
        errseq_t                        f_sb_err;
#ifdef CONFIG_EPOLL
        struct hlist_head               *f_ep;
#endif
        union {
                struct callback_head    f_task_work;
                struct llist_node       f_llist;
                struct file_ra_state    f_ra;
                freeptr_t               f_freeptr;
        };
        /* --- cacheline 3 boundary (192 bytes) --- */
} __randomize_layout
  __attribute__((aligned(4)));  /* lest something weird
```

文件对象file

- f_ref 表示文件的引用计数，用于管理文件对象的生命周期；
- f_lock 是一个自旋锁，用于保护 f_ep、f_flags 等字段的并发访问；
- f_mode 存储了 FMODE_* 标志，描述了文件的打开模式和操作特性；
- f_op 指向文件操作集，定义了对该文件进行读写、ioctl 等操作的函数；
- f_mapping 是文件内容的地址空间，管理文件内容的缓存和内存映射；
- private_data 存储文件系统或驱动程序的特定数据，为文件提供额外信息；
- f_inode 缓存了与该文件关联的 inode，便于快速访问文件元数据；
- f_path 描述了文件在文件系统中的路径信息，构成文件定位的关键；
- f_pos 记录了文件当前的读写位置，支持文件数据的顺序访问。

文件对象file

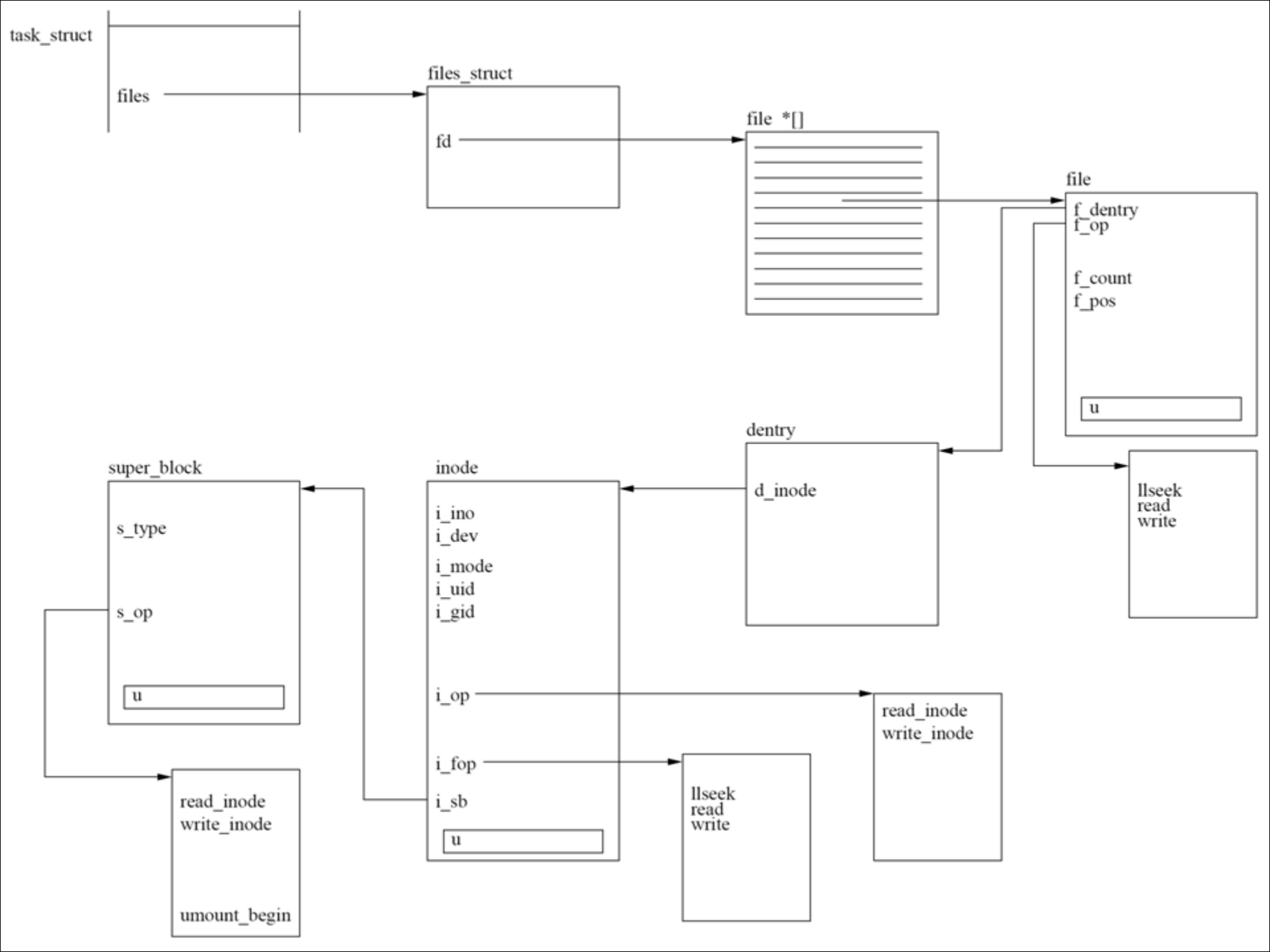
字段名	实例化值/行为描述	关联场景（以 `open("/etc/passwd", O_RDONLY)` 为例）
`f_ref`	引用计数初始为 `1`，表示当前有一个进程持有该文件对象。若其他进程打开同一文件，计数递增。	用户调用 `open()` 时计数加 1，调用 `close()` 时减 1。
`f_mode`	`FMODE_READ`（只读）。	用户尝试写入文件时，VFS 检查 `f_mode` 是否包含 `FMODE_WRITE`，若否则拒绝。
`f_mapping`	指向文件的地址空间（`address_space`），与 `inode->i_mapping` 相同，管理页缓存和内存映射。	用户调用 `mmap()` 时，内核通过 `f_mapping` 建立文件到内存的映射。
`private_data`	空。	`/etc/passwd` 是普通文件，此字段为空。
`f_inode`	指向关联的 `inode`（即 `/etc/passwd` 的 inode，`i_ino=74191564`）。	通过 `f_inode` 访问文件的元数据（如 `i_size=3762`）。
`f_path`	包含两个成员：`dentry`：指向 `/etc/passwd` 的目录项（`dentry`）；`vfsmount`：文件所在挂载点的信息。	用于生成绝对路径（如 `/proc/self/fd` 中的符号链接）。
`f_pos`	当前读写位置偏移量（单位：字节）。初始为 `0`，每次 `read()` 或 `lseek()` 后更新。	用户调用 `read(fd, buf, 1024)` 后，`f_pos` 变为 `1024`。

文件对象操作file_operations

```
struct file_operations {
        struct module *owner;
        fop_flags_t fop_flags;
        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 (*iopoll)(struct kiocb *kiocb, struct io_comp_batch *,
                        unsigned int flags);
        int (*iterate_shared) (struct file *, struct dir_context *);
        __poll_t (*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 (*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 (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        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);
        void (*splice_eof)(struct file *file);
        int (*setlease)(struct file *, int, struct file_lease **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,
```

文件对象操作file_operations

- llseek：用于调整文件的读写偏移位置；
- read：从文件中读取数据到用户空间；
- write：将数据从用户空间写入文件；
- open：在文件打开时初始化文件对象及相关状态；
- release：在文件关闭时进行清理和资源释放；
- mmap：实现文件内容的内存映射，允许进程直接访问文件数据；
- unlocked_ioctl：用于执行设备或文件的控制命令操作；
- poll：支持事件轮询，使应用能够检测文件状态的变化；
- fsync：确保文件数据及元数据被同步写入底层存储设备。


task_struct
files
files_struct
fd
file *[]
file
f_dentry
f_op
f_count
f_pos
u
dentry
d_inode
llseek
read
write
super_block
s_type
s_op
u
inode
i_ino
i_dev
i_mode
i_uid
i_gid
i_op
i_fop
i_sb
u
read_inode
write_inode
read_inode
write_inode
umount_begin
llseek
read
write

文件管理和操作

- 对于系统中打开的文件，主要从两个方面进行管理，一是由系统通过系统打开文件表进行统一管理，另一是由进程通过私有数据结构进行管理。文件打开后要进行各种操作，VFS提供了面向文件操作的统一接口。
- 系统打开文件表:
  - Linux系统内核把所有进程打开的文件集中管理，把它们组成"系统打开文件表"。
  - 系统打开文件表是一个双向链表，它的每个表项(节点)是一个file结构。
  - 进程打开一个文件就建立一个file结构，并把它加入到系统打开文件链表中。
  - 全局变量first_file指向系统打开文件表的表头。

文件管理和操作

- 进程的文件管理：
  - 对于一个进程打开的所有文件，由进程的两个私有结构进行管理。
    - fs_struct结构体记录了文件系统根目录和当前目录。
    - files_struct结构体包含了进程的打开文件表。

```
struct fs_struct {
        int users;
        spinlock_t lock;
        seqcount_spinlock_t seq;
        int umask;
        int in_exec;
        struct path root, pwd;
} __randomize_layout;
```

```
struct files_struct {
  /*
   * read mostly part
   */
        atomic_t count;
        bool resize_in_progress;
        wait_queue_head_t resize_wait;

        struct fdtable __rcu *fdt;
        struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
        spinlock_t file_lock ____cacheline_aligned_in_smp;
        unsigned int next_fd;
        unsigned long close_on_exec_init[1];
        unsigned long open_fds_init[1];
        unsigned long full_fds_bits_init[1];
        struct file __rcu * fd_array[NR_OPEN_DEFAULT];
};
```

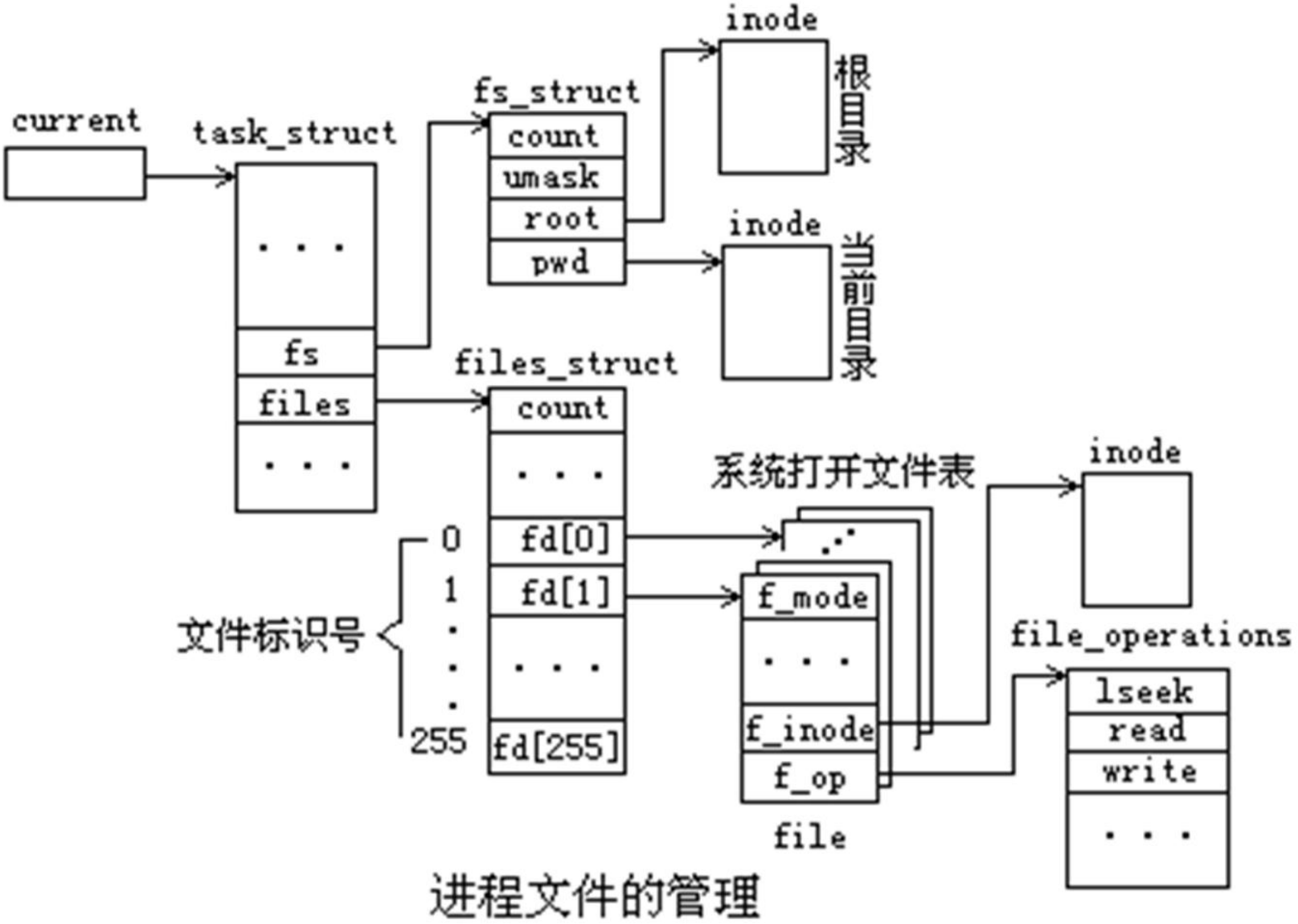
文件管理和操作

- count 是一个原子计数器，用于管理整个 files_struct 的引用计数；fdt 是一个受 RCU 保护的指针，指向当前正在使用的文件描述符表 fdtable；
- fdtab 是内嵌的初始文件描述符表，在文件描述符数量较少时直接使用，避免动态分配开销；
- file_lock 是自旋锁，用于保护诸如 next_fd 和 fd_array 等字段在 SMP 环境下的并发修改；
- next_fd 指示下一个可分配的文件描述符索引，帮助快速定位空闲位置；
- close_on_exec_init 是一个位图数组，记录哪些文件描述符在执行 exec 系统调用时需要关闭；
- open_fds_init 是一个位图数组，用于标记当前处于打开状态的文件描述符；
- full_fds_bits_init 是一个位图数组，记录文件描述符表中所有已分配的位信息，以便检测表是否已满；
- fd_array 是一个默认大小的数组，存储所有打开文件的指针，实现文件描述符与具体文件对象之间的映射。

文件管理和操作

- fd_array每个数组元素对应一个文件描述符，而文件描述符本身就是fd_array 的下标。
- 当进程通过文件名打开一个文件后，系统会在 fd_array 中找到一个空闲位置，将对应的 file 结构指针存入该位置，并将该下标作为文件描述符返回，此后进程对文件的操作均通过这个数字来识别文件，而不再使用文件名。
- 系统在启动时预先分配了文件描述符 0、1、2，分别对应标准输入、标准输出和标准错误输出设备。
- 当一个进程通过 fork() 创建子进程时，父子进程共享同一个打开文件表，这意味着它们在相同下标上的 fd_array 元素指向相同的 file 结构，此时 file 结构中的 f_count 引用计数会增加，以反映多个进程对同一文件的引用。
- 另外，一个文件可以被同一进程多次打开，每次打开都会在打开文件表中分配一个新的 file 结构并占用一个新的 fd_array 项目，虽然这些 file 结构是独立的，但它们内部的 f_inode 指针均指向同一个 inode，从而保持对该文件元数据的统一管理。

文件管理和操作



进程文件的管理

文件的open()操作

- 在 Linux 内核中，文件的 open 操作始于 sys_open() 系统调用，该调用接收用户传入的文件路径和访问标志。

1. 内核通过 getname() 将用户空间中的文件路径复制到内核空间，确保路径数据的有效性。
2. 系统利用 get_unused_fd() 在当前进程的打开文件表（current->files->fd 数组）中寻找一个空闲的文件描述符，并将其下标保存到局部变量中。
3. 内核调用 filp_open() 来完成文件的打开工作，这一过程分为两步：
    1. 第一步调用 open_namei() 进行路径解析，通过解析得到目标文件对应的 dentry 对象以及关联的 inode 对象；
    2. 第二步则通过 dentry_open() 为该 dentry 分配并初始化 file 对象，将 file 对象的 f_dentry 成员指向解析出的 dentry。
4. 最后，fd_install() 函数将新创建的 file 对象安装到当前进程的打开文件表中，即 current->files->fd[fd] 被赋值为该 file 对象，最终返回文件描述符，从此进程对文件的识别不再依赖文件名，而是直接使用文件描述符进行后续操作。

文件的open()操作

- open_namei() 是完成文件 open 操作中的关键步骤，它首先确定从哪一个 dentry 作为起点进行路径解析，这取决于所提供的文件路径是相对路径还是绝对路径。
- 通过 path_walk()（实际上由 link_path_walk() 实现）进入一个循环，每次解析路径的一个组件。
- 在这个循环中，内核处理诸如“.”（表示当前目录，直接跳过）和“..”（表示父目录，需要更新至父目录的 dentry）的特殊情况。
  - 内核首先调用 cache_lookup() 检查当前路径组件是否已经存在于 dentry 缓存中，如果存在，则直接使用缓存中的 dentry，加快解析速度；否则，会分配新的 dentry 对象，并调用 real_lookup() 从磁盘读取该目录项的信息，real_lookup() 会利用父目录 inode 中定义的 lookup() 方法读取目标 dentry 对应的 inode 数据，同时在必要时处理挂载点和符号链接，将解析推进到安装设备的根节点或连接目标。
  - 最终，当所有路径组件都被正确解析后，open_namei() 完成对目标 dentry 和其关联 inode 的定位，为 dentry_open() 初始化 file 对象提供了充分的信息，整个过程确保了文件系统能够高效地将用户提供的路径名转换为内核中用于后续 I/O 操作的 file 对象。

ext4文件系统

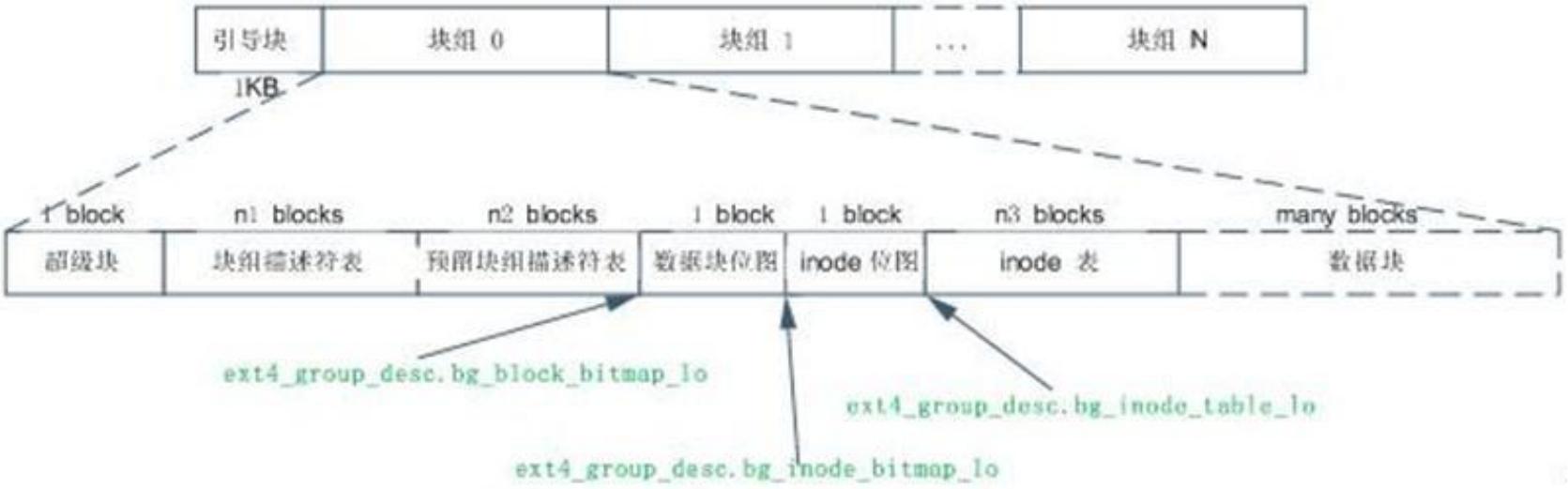
- ext4 文件系统采用先进的数据结构和分配算法，专为高性能和大容量存储设计；
- 支持的分区容量可达 1 EB，单个文件的最大尺寸高达 16 TB；
- 支持长文件名，通常最长为 255 个字符，并支持丰富的扩展属性；
- 引入了 extents 机制，用以替代传统块映射，显著降低文件碎片并提升大文件的读写效率；
- 采用延迟分配、多块分配等优化技术，进一步提高了磁盘 I/O 性能；
- 利用日志校验和日志恢复机制，确保数据完整性和快速恢复；
- 支持在线碎片整理、快速挂载和热插拔等高级特性，适应动态存储需求；
- 同时，ext4 使用专用文件记录文件系统状态和错误信息，以便在系统启动时进行一致性检查；
- 综合性能、可靠性与灵活性，ext4 已成为现代 Linux 系统中主流的大容量存储解决方案。

ext4文件系统在磁盘上的物理布局

- 在 ext4 中，磁盘的物理布局基本上沿袭了 ext2 的分组设计，但增加了灵活性和扩展性。
- 磁盘的起始区域保留用于引导加载程序，紧接着在固定偏移（通常为 1024 字节处）存放主超级块和备份超级块（根据稀疏超级块策略，并非每个组都保存备份），随后是组描述符表，它记录了各组的位置信息。
- 磁盘被划分为多个大小相同的块组，而 ext4 引入了“flex block groups”的概念，将若干个**块组**组织在一起以减少元数据的复制开销。
- 每个块组内通常包含数据块位图、inode 位图、inode 表（其中 inode 采用了 extents 结构来高效管理大文件的块映射）以及实际的数据块区域。
- 此外，如果启用了日志功能，还会有专门区域存放日志数据，这些设计共同确保了 ext4 在管理海量存储、提高性能和维护数据完整性方面的优势。

ext4体系结构

- **超级块**是文件系统中最重要的结构，它描述了整个文件系统的基本信息；
- **组描述符**记录了各块组的信息，如块组中空闲块和空闲 inode 的数量；
- 每个块组都有一个**数据块位图**，其大小为一个块，每一位对应组中的一个数据块，0 表示可用，1 表示已使用；每个块组同时配备一个 inode 位图，用来表示该块组中 inode 表空间的占用情况；
- **inode 表**存储了文件和目录的 inode 数据，每个文件或目录均由一个 inode 进行表示。



ext4超级块

- 超级块同样是用来描述整个文件系统整体信息的重要数据结构，它详细记录了文件系统目录和文件的静态分布情况，以及各组成结构（如块组、inode 表等）的尺寸、数量等参数。
- 在 ext4 中，超级块位于每个块组的最前端，每个块组中的超级块内容基本相同，但在系统运行时，内核只需将块组0中的超级块读入内存，其他块组的超级块则作为备份存在，这样既保证了数据的冗余性，也便于在出现问题时恢复文件系统。
- 在 Linux 中，ext4 超级块结构定义为 ext4_super_block。

ext4超级块

```
struct ext4_super_block {
/*00*/    __le32  s_inodes_count;         /* Inodes count */
          __le32  s_blocks_count_lo;      /* Blocks count */
          __le32  s_r_blocks_count_lo;    /* Reserved blocks count */
          __le32  s_free_blocks_count_lo; /* Free blocks count */
/*10*/    __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_cluster_size;     /* Allocation cluster size */
/*20*/    __le32  s_blocks_per_group;     /* # Blocks per group */
          __le32  s_clusters_per_group;   /* # Clusters per group */
          __le32  s_inodes_per_group;     /* # Inodes per group */
          __le32  s_mtime;                /* Mount time */
/*30*/    __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 */
/*40*/    __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 */
/*50*/    __le16  s_def_resuid;           /* Default uid for reserved blocks */
          __le16  s_def_resgid;           /* Default gid for reserved blocks */
```

字段名 (偏移量)	实例化值（十六进制/十进制）	说明
`s_inodes_count` (0x00)	`0x0000C800`（51,200）	文件系统可分配的总 inode 数量，与 `mkfs.ext4` 的 `-N` 参数相关。
`s_blocks_count_lo` (0x04)	`0x00100000`（1,048,576）	文件系统总块数（低 32 位）。若块大小为 4KB，总容量为 `1,048,576 * 4KB = 4GB`。
`s_r_blocks_count_lo` (0x08)	`0x00001000`（4,096）	保留块数（用于 root 用户紧急恢复），通常为总块数的 5%。
`s_free_blocks_count_lo` (0x0C)	`0x00020000`（131,072）	当前空闲块数（低 32 位），表示剩余可用空间约 `131,072 * 4KB = 512MB`。
`s_free_inodes_count` (0x10)	`0x00002000`（8,192）	当前空闲 inode 数，若 `s_inodes_count=51,200`，则已分配约 `51,200 - 8,192 = 43,008` 个 inode。
`s_first_data_block` (0x14)	`0x00000001`（1）	第一个数据块的编号（0 或 1），现代 ext4 通常为 1（块 0 保留给引导扇区）。
`s_log_block_size` (0x18)	`0x00000002`（2）	块大小的对数表示：`块大小 = 1024 << s_log_block_size = 4096` 字节。
`s_log_cluster_size` (0x1C)	`0x00000002`（2）	簇大小的对数（用于 bigalloc 特性），若未启用则与块大小相同，即簇大小也为 4KB。
`s_blocks_per_group` (0x20)	`0x00008000`（32,768）	每个块组包含的块数，典型默认值。块组总数 = `s_blocks_count_lo / s_blocks_per_group = 32`。
`s_clusters_per_group` (0x24)	`0x00008000`（32,768）	每个块组的簇数，与 `s_blocks_per_group` 一致（未启用 bigalloc）。
`s_inodes_per_group` (0x28)	`0x00002000`（8,192）	每个块组的 inode 数，块组总数 = `s_inodes_count / s_inodes_per_group = 6`。
`s_mnt_count` (0x34)	`0x0012`（18）	文件系统挂载次数，超过 `s_max_mnt_count` 时触发 `fsck`。
`s_max_mnt_count` (0x36)	`0xFFFF`（-1）	最大挂载次数限制，`-1` 表示无限制。
`s_magic` (0x38)	`0xEF53`	ext4 文件系统魔数标识，固定为 `0xEF53`。
`s_state` (0x3A)	`0x0001`（EXT4_VALID_FS）	文件系统状态：`1` 表示已干净卸载，`0` 表示需要检查。
`s_errors` (0x3C)	`0x0001`（EXT4_ERRORS_CONTINUE）	错误处理策略：`1` 表示忽略错误继续运行，`2` 表示挂载为只读，`3` 触发内核 panic。
`s_checkinterval` (0x44)	`0x000F4240`（1,000,000 秒 ≈ 11.5 天）	两次强制 `fsck` 检查的最大间隔时间。
`s_creator_os` (0x48)	`0x00000000`（EXT4_OS_LINUX）	创建文件系统的操作系统，`0` 表示 Linux。

ext4组描述符

- 每个组都有自己的描述符，用来描述一个块组的有关信息，内核用ext4_group_desc描述。

```
struct ext4_group_desc
{
        __le32  bg_block_bitmap_lo;     /* Blocks bitmap block */
        __le32  bg_inode_bitmap_lo;     /* Inodes bitmap block */
        __le32  bg_inode_table_lo;      /* Inodes table block */
        __le16  bg_free_blocks_count_lo;/* Free blocks count */
        __le16  bg_free_inodes_count_lo;/* Free inodes count */
        __le16  bg_used_dirs_count_lo;  /* Directories count */
        __le16  bg_flags;               /* EXT4_BG_flags (INODE_UNINIT, etc) */
        __le32  bg_exclude_bitmap_lo;   /* Exclude bitmap for snapshots */
        __le16  bg_block_bitmap_csum_lo;/* crc32c(s_uuid+grp_num+bbitmap) LE */
        __le16  bg_inode_bitmap_csum_lo;/* crc32c(s_uuid+grp_num+ibitmap) LE */
        __le16  bg_itable_unused_lo;    /* Unused inodes count */
        __le16  bg_checksum;            /* crc16(sb_uuid+group+desc) */
        __le32  bg_block_bitmap_hi;     /* Blocks bitmap block MSB */
        __le32  bg_inode_bitmap_hi;     /* Inodes bitmap block MSB */
        __le32  bg_inode_table_hi;      /* Inodes table block MSB */
        __le16  bg_free_blocks_count_hi;/* Free blocks count MSB */
        __le16  bg_free_inodes_count_hi;/* Free inodes count MSB */
        __le16  bg_used_dirs_count_hi;  /* Directories count MSB */
        __le16  bg_itable_unused_hi;    /* Unused inodes count MSB */
        __le32  bg_exclude_bitmap_hi;   /* Exclude bitmap block MSB */
        __le16  bg_block_bitmap_csum_hi;/* crc32c(s_uuid+grp_num+bbitmap) BE */
        __le16  bg_inode_bitmap_csum_hi;/* crc32c(s_uuid+grp_num+ibitmap) BE */
        __u32   bg_reserved;
};
```

inode索引节点

- 在 ext4 文件系统中，inode 的默认大小通常为 256 字节，inode 按顺序存储在 inode 表中，该表由一系列连续的块构成。
- inode 表中第一个块的块号存放在块组描述符的 bg_inode_table 字段中，所有 inode 的大小均为 256 字节。因此，一个 1024 字节的块可以存放 4 个 inode，而一个 4096 字节的块可以存放 16 个 inode。
- 要计算 inode 表占用了多少块，只需将一个块组中 inode 的总数（记录在超级块的 s_inodes_per_group 字段中）除以每块中可以容纳的 inode 数量即可。

ext4_inode

```
struct ext4_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size_lo;      /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Inode Change 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_lo;    /* Blocks count */
        __le32  i_flags;        /* File flags */
        union {
                struct {
                        __le32  l_i_version;
                } linux1;
                struct {
                        __u32  h_i_translator;
                } hurd1;
                struct {
                        __u32  m_i_reserved1;
                } masix1;
        } osd1;                         /* OS dependent 1 */
        __le32  i_block[EXT4_N_BLOCKS];/* Pointers to blocks */
        __le32  i_generation;   /* File version (for NFS) */
        __le32  i_file_acl_lo;  /* File ACL */
        __le32  i_size_high;
        __le32  i_obso_faddr;   /* Obsoleted fragment address */
```

ext4_inode

- i_mode 字段编码了文件的类型和权限。例如，普通文件、目录、字符设备、块设备、命名管道、套接字、符号链接及未知类型分别对应特定的编码值。
- 同时，ext4 中继承了 ext2 的数据块指针数组（EXT4_N_BLOCKS，默认值通常为 15），用于存放文件数据所在磁盘块的编号，不过在 ext4 中对于大文件更常采用 extents 机制以提高效率。
  - 不同类型的文件对这个数组的用途也有所不同：对于设备文件，inode 内部的元数据足以记录所有必要信息，不需要额外的数据块；
  - 而对于目录，数据块中存储着目录项信息，每个数据块中包含若干个 ext4_dir_entry_2 结构，用来依次描述该目录下各文件的信息。

ext4_inode

字段名	实例化值（十六进制/十进制）	说明
`i_mode`	`0x81A4`（`S_IFREG | 0644`）	普通文件（`S_IFREG`），权限 `rw-r--r--`。
`i_uid`	`0x0000`（低 16 位 UID=0，root）	结合 `osd2.linux2.l_i_uid_high` 可支持 32 位 UID。
`i_size_lo`	`0x00000EB2`（3,762 字节）	文件逻辑大小（对应 `stat` 中的 `大小: 3762`）。
`i_dtime`	`0x00000000`	文件未删除，删除时间未设置。
`i_gid`	`0x0000`（低 16 位 GID=0，root）	结合 `osd2.linux2.l_i_gid_high` 可支持 32 位 GID。
`i_links_count`	`0x0001`（硬链接数为 1）	`/etc/passwd` 未被其他目录硬链接。
`i_blocks_lo`	`0x00000008`（8 个 512 字节块）	文件占用磁盘空间：`8 * 512 = 4096` 字节（对应 `stat` 中的 `块: 8`）。
`i_flags`	`0x00000000`	默认无扩展属性（如加密、压缩等）。
`i_size_high`	`0x00000000`	文件大小未超过 4GB，无需使用高 32 位。
`i_ctime_extra`	`0x0000027C`（纳秒=620，epoch=0）	变更时间的纳秒部分：`620 << 2 = 2480 ns`。
`i_mtime_extra`	`0x0000015A`（纳秒=350，epoch=0）	修改时间的纳秒部分：`350 << 2 = 1400 ns`。
`i_atime_extra`	`0x00000384`（纳秒=900，epoch=0）	访问时间的纳秒部分：`900 << 2 = 3600 ns`。
`i_checksum_hi`	`0x12AB`	inode 校验和高 16 位（CRC32C 算法生成）。
`osd2.linux2.l_i_checksum_lo`	`0xCD34`	inode 校验和低 16 位，与 `i_checksum_hi` 组合为完整校验和。

ext4_dir_entry_2

```
struct ext4_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;              /* See file type macros EXT4_FT_* below */
        char    name[EXT4_NAME_LEN];    /* File name */
};
```

- inode 字段（32 位小端整数）存储了目录项对应的 inode 编号；
- rec_len 字段（16 位小端整数）表示该目录项记录的总长度；
- name_len 字段占 1 字节，用于指示文件名的长度；
- file_type 字段占 1 字节，记录文件的类型（参见 EXT4_FT_* 宏定义）；
- name 数组用于存放实际的文件名。

proc文件系统

- proc 文件系统是 Linux 中的特殊文件系统，提供给用户一个可以了解内核内部工作过程的可读窗口，在运行时访问内核内部数据结构、改变内核设置的机制。
  - 保存系统当前工作的特殊数据，但并不存在于任何物理设备中；
  - 对其进行读写时，才根据系统中的相关信息即时生成；或映射到系统中的变量或数据结构；
  - proc 被称为“伪文件系统”；
  - 其挂接目录点固定为/proc；
  - man proc 进行了详细说明。

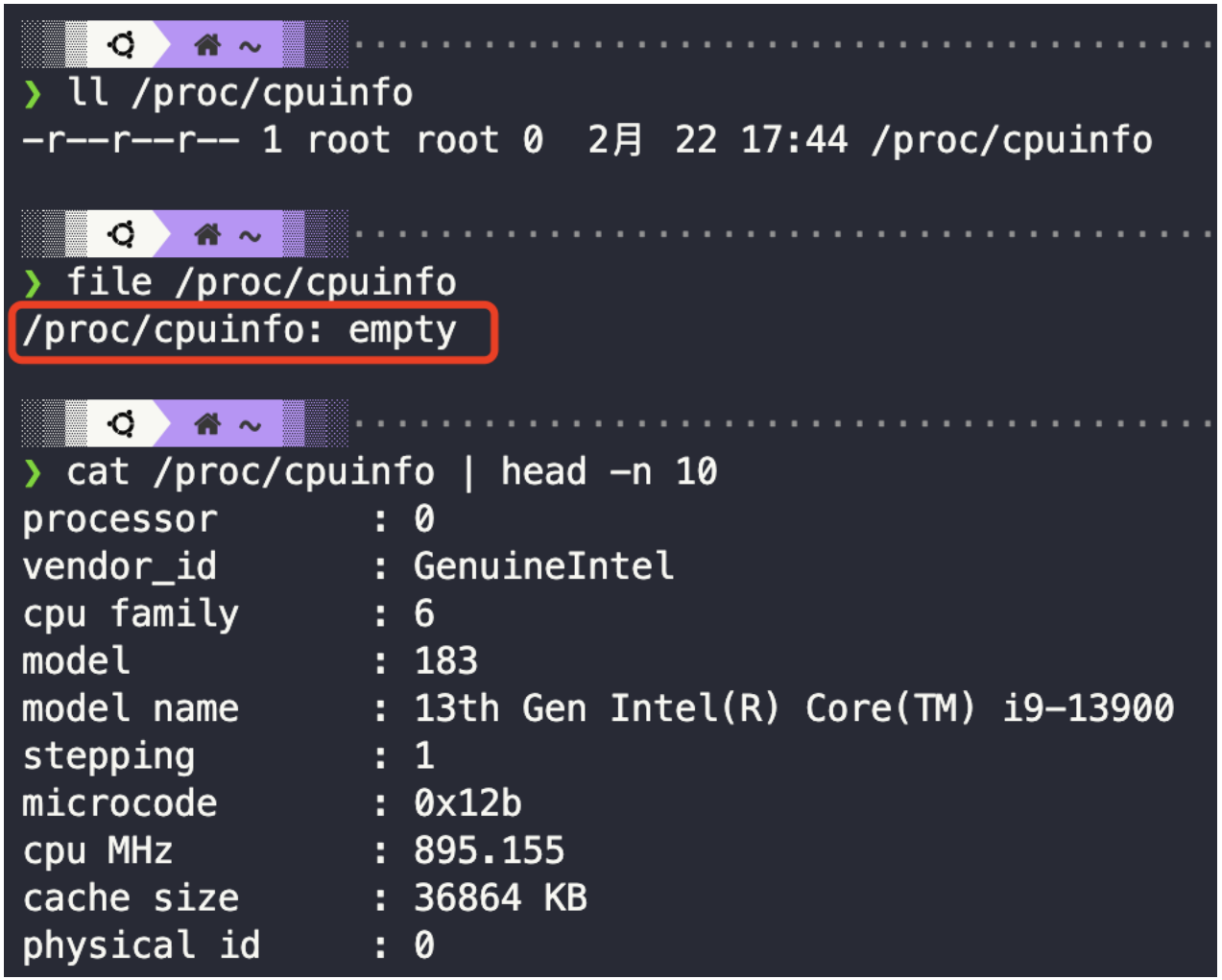
proc文件系统

- /proc 的文件可以用于访问有关内核的状态、计算机的属性、正在运行的进程的状态等信息。大部分 /proc 中的文件和目录提供系统物理环境最新的信息。
- 尽管 /proc 中的文件是虚拟的，但它们仍可以使用任何文件编辑器或像'more', 'less'或 'cat'这样的程序来查看。当编辑程序试图打开一个虚拟文件时，这个文件就通过内核中的信息被凭空地 (on the fly) 创建了 。

proc文件系统

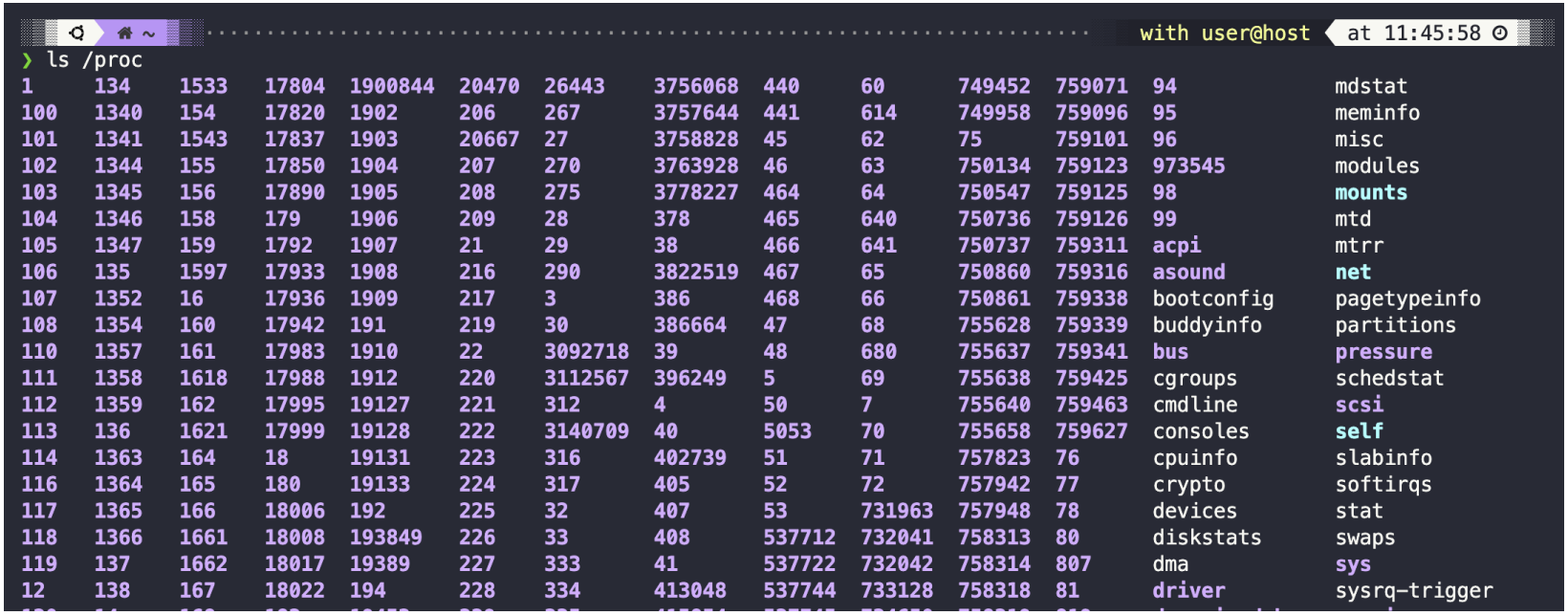
- proc 文件系统可以被用于收集有用的关于系统和运行中的内核的信息。下面是一些重要的文件：
  - /proc/cpuinfo - CPU 的信息 (型号, 家族, 缓存大小等)
  - /proc/meminfo - 物理内存、交换空间等的信息
  - /proc/mounts - 已加载的文件系统的列表
  - /proc/devices - 可用设备的列表
  - /proc/filesystems - 被支持的文件系统
  - /proc/modules - 已加载的模块
  - /proc/version - 内核版本
  - /proc/cmdline - 系统启动时输入的内核命令行参数
- proc 中的文件远不止上面列出的这么多。想要进一步了解，可以对 /proc 的每一个文件都'more'一下。

proc文件系统


~
ll /proc/cpuinfo
-r--r--r-- 1 root root 0 2月 22 17:44 /proc/cpuinfo
~
file /proc/cpuinfo
/proc/cpuinfo: empty
~
cat /proc/cpuinfo | head -n 10
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 183
model name : 13th Gen Intel(R) Core(TM) i9-13900
stepping : 1
microcode : 0x12b
cpu MHz : 895.155
cache size : 36864 KB
physical id : 0

proc文件系统

- /proc 文件系统可以用于获取运行中的进程的信息。在 /proc 中有一些编号的子目录。每个编号的目录对应一个进程 id (PID)。这样，每一个运行中的进程 /proc 中都有一个用它的 PID 命名的目录。这些子目录中包含可以提供有关进程的状态和环境的重要细节信息的文件。让我们试着查找一个运行中的进程，见下页。



proc文件系统

- 文件
  - “cmdline” 包含启动进程时调用的命令行。
  - "cpu"仅在运行 SMP 内核时出现，里面是按 CPU 划分的进程时间。
  - “cwd”是指向进程当前工作目录的符号链接，
  - “envir” 进程的环境变量。
  - “exe”指向运行的进程的可执行程序，
- 目录
  - “fd”包含指向进程使用的文件描述符的链接。
  - “root”指向被这个进程看作是根目录的目录 (通常是“/”)。
  - “status” 是进程的状态信息，包括启动进程的用户的用户ID (UID) 和组ID(GID)，父进程ID (PPID)，还有进程当前的状态，比如“Sleelping”和“Running”。每个进程的目录都有几个符号链接，

```
❯ sudo ls /proc/741617
arch_status  comm                fd               limits     mountstats    pagemap      sessionid     status          wchan
attr         coredump_filter     fdinfo           loginuid   net           patch_state  setgroups     syscall
autogroup    cpu_resctrl_groups  gid_map          map_files  ns            personality  smaps         task
auxv         cpuset              io               maps       numa_maps     projid_map   smaps_rollup  timens_offsets
cgroup       cwd                 ksm_merging_pages  mem      oom_adj       root         stack         timers
clear_refs   environ             ksm_stat         mountinfo  oom_score     sched        stat          timerslack_ns
cmdline      exe                 latency          mounts     oom_score_adj  schedstat   statm         uid_map
```

proc文件系统

- 上面讨论的大部分 /proc 的文件是只读的。而实际上 /proc 文件系统通过 /proc 中可读写的文件提供了对内核的交互机制。写这些文件可以改变内核的状态，因而要慎重改动这些文件。
- /proc/sys 目录存放所有可读写的文件的目录，可以被用于改变内核行为。
- /proc/sys/kernel - 这个目录包含反通用内核行为的信息。/proc/sys/kernel/{domainname, hostname} 存放着机器/网络的域名和主机名。这些文件可以用于修改这些名字。

```
[root@myose /proc]# hostname
myose
[root@myose /proc]# cat /proc/sys/kernel/hostname
myose
[root@myose /proc]# echo NEWose > /proc/sys/kernel/hostname
[root@myose /proc]# hostname
NEWose
[root@myose /proc]#
```

proc文件系统

- 这样，通过修改 /proc 文件系统中的文件，可以修改主机名或者文件系统的可分配文件句柄的最大数值等等。很多其他可配置的文件存在于 /proc/sys/kernel/。这里不可能列出所有这些文件，同学们可以自己去这个目录查看以得到更多细节信息。

```
[root@myose /proc]# cat /proc/sys/fs/file-max
4096
[root@myose /proc]# echo 8192 > /proc/sys/fs/file-max
[root@myose /proc]# cat /proc/sys/fs/file-max
8192
[root@myose /proc]#
```

proc文件系统的编程接口

- 前面学习了proc文件系统的基本概念。本次实验将编写一个内核模块。通过加载模块，在/proc目录下增加若干个文件，用户对文件的读写都由模块进行处理。
- /proc目录下的文件属于一种特殊的文件，必须用特定的方法创建和删除
- proc **文件系统的编程接口比较好记，大部分函数是VFS函数名前面加上一个”proc_”**
  - 创建目录函数proc_mkdir()；
  - 创建符号链接函数proc_symlink()；
  - 创建设备文件函数proc_mknod()；

proc文件系统的编程接口

- 介绍内核函数之前。先来了解proc文件系统编程最主要的数据结构—— proc_dir_entry

```
struct proc_dir_entry {
  unsigned short low_ino;
  unsigned short namelen;
  const char *name;
  mode_t mode;
  nlink_t nlink;
  uid_t uid;
  gid_t gid;
  unsigned long size;
  struct inode_operations *proc_iops;
  struct file_operations *proc_fops;
  get_info_t *get_info;
  struct module *owner;
  struct proc_dir_entry *next,
*parent, *subdir;
  void *data;
  read_proc_t *read_proc;
  write_proc_t *write_proc;
  atomic_t count; /*使用计数*/
  int deleted;       /*删除标志*/
  kdev_t rdev;
};
```

proc文件系统的编程接口

- **每一个这样的数据结构代表了一个节点，也就是一个proc文件。其中很多结构成员的意义和普通文件的一样。编程中用到的成员并不多。几个常用到的成员如下：**
  - name：　　节点的名称，也就是该proc文件的名称
  - mode：　　文件的类型和权限
  - nlink：　　该文件的链接数
  - read_proc：　读操作函数
  - write_proc：写操作函数
  - owner：　　该文件的拥有者模块

proc文件系统的编程接口

- 下面介绍几个内核函数。通过这些函数，可以请求内核在proc文件系统中创建或者删除文件或目录。
- 要注意这些函数都是内核函数，只能在核心态被调用，需要编写一个内核模块去调用它们。

proc文件系统的编程接口

- 创建文件create_proc_entry()

  struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode, struct proc_dir_entry *parent)

- 该函数将创建一个proc文件，文件名为name，文件类型和访问权限为mode，父目录为parent。
- 如果想在proc文件系统的根目录下创建，则制定参数parent为NULL。
- 和普通文件不同的是，proc文件系统允许在同一个目录下创建多个同名的文件和子目录
- 创建的文件和目录不能用常规文件系统的rm或rmdir删除

proc文件系统的编程接口

- 创建只读文件create_proc_read_entry()

  struct proc_dir_entry *create_proc_read_entry(const char *name, mode_t mode, struct proc_dir_entry *base, read_proc_t *read_proc, void *date)

- 该函数将创建一个只读的proc文件，其实它只是简单地调用create_proc_entry，并将返回结构的read_proc域的值置为read_proc，data域的值置为data。

proc文件系统的编程接口

- 创建目录create_mkdir()

  struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent)

- 该函数将创建一个目录，父目录为parent。

/proc/edsionte_procfs/foo:

- struct proc_dir_entry *example_dir = proc_mkdir(“edsionte_procfs”, NULL);
- struct proc_dir_entry *foo_file = create_proc_entry("foo", 0644, example_dir);

```
static int proc_read_foobar(char *page, char **start,
off_t off, int count, int *eof, void *data)
{
    int len;
    struct fb_data_t *fb_data = (struct fb_data_t *)data;
    //将fb_data的数据写入page
    len = sprintf(page, "%s = %s\n", fb_data->name, fb_data-
    >value);
    return len;
    }
```

```
static int proc_write_foobar(struct file *file, const char *buffer,
unsigned long count, void *data)
{
    int len;
    struct fb_data_t *fb_data = (struct fb_data_t *)data;
    if (count > FOOBAR_LEN)
    len = FOOBAR_LEN;
    else
    len = count;
    //写函数的核心语句，将用户态的buffer写入内核态的value中
    if (copy_from_user(fb_data->value, buffer, len))
    return -EFAULT;
    fb_data->value[len] = '\0';
    return len;
}
```

proc文件系统的编程接口

- 删除节点（文件或者目录）remove_proc_entry()

  void remove_proc_entry (const char *name, struct proc_dir_entry *parent)

- 该函数将删除一个proc节点（按文件名删除）。

proc文件系统的编程接口

- 创建符号链接proc_symlink()

  struct proc_dir_entry *proc_symlink(const char *name, struct proc_dir_entry *parent, char *dest)

- 该函数在parent目录下创建一个名字为name的符号链接文件，链接的目标是dest。

proc文件系统的编程接口

- 创建设备文件proc_mknod()

  struct proc_dir_entry *proc_mknod(const char *name, mode_t mode, struct proc_dir_entry *parent, kdev_t *rdev)

- 该函数在parent目录下创建一个名字为name的设备文件，文件类型和权限为mode，设备号为rdev。

proc文件系统的编程接口

- 以上五个创建节点的函数在内核中的实现流程：
  - 通过proc_create为结构申请空间，并进行一些初始化工作。
  - proc_register则进一步填写结构中的域。并完成注册工作

- 删除节点的函数在内核中的实现流程：
  - 则是先调用clear_bit和proc_kill_inodes，注销inode结构，如果引用数为0，则调用free_proc_entry释放结构对应的空间；否则置一个删除标志，不释放空间

proc文件系统的编程接口

- 以上函数只能创建一个文件，要想使创建的文件发挥作用，还有两个域的值需要填写，它们是read_proc和write_proc。
- 该两个函数都是回调函数，当对文件进行读写时，系统会自动调用相应的回调函数。

```
int (*read_proc) (char *page, char **start, off_t off, int count, int *eof,
    void *data)

int (*write_proc) (struct file *file, const char *buffer, unsigned long
    count, void *data)
```

实验：添加一个文件系统

- 实验中用到的命令dd：用指定大小的块拷贝一个文件，并在拷贝的同时进程指定的转换。
  - 常用参数:
    - if=输入文件(或设备名称)
    - of=输出文件(或设备名称)
    - bs =bytes 同时设置读/写缓冲区的字节数(等于设置ibs和obs)count=blocks 只拷贝输入的blocks块
    - conv=ucase把字母由小写转换为大写
    - conv=lcase 把字母由大写转换为小写
  - 例如：dd if=/dev/zero of=myfs bs=1M count=1。该命令从/dev/zero 读取 1MB 的零值数据，并写入文件 myfs。
- 实验方案已上传：http://10.10.21.30/linux-kernel/practice_kern/-/tree/main/AddMyEXT4

实验：向proc文件系统中添加目录和文件

- 参考：http://10.10.21.30/linux-kernel/practice_kern/-/tree/main/CreateProc?ref_type=heads。