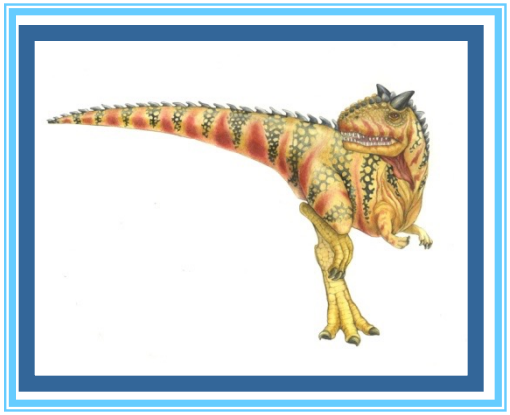
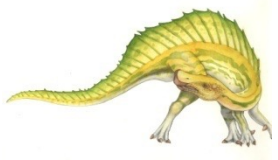
文件系统

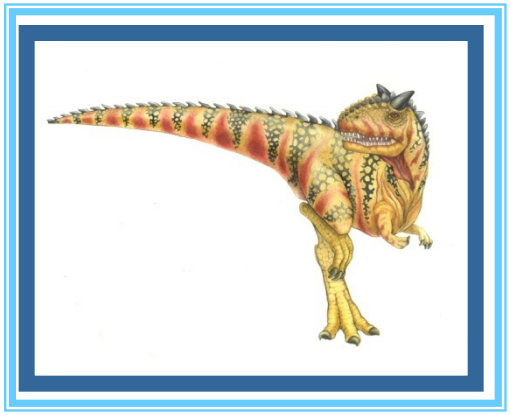


本章内容

- Linux文件系统概述
- 虚拟文件系统VFS
- 物理文件系统 EXT2
- 文件open、read实现

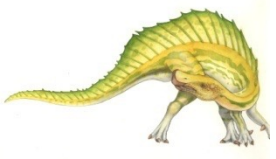


Linux文件系统概述

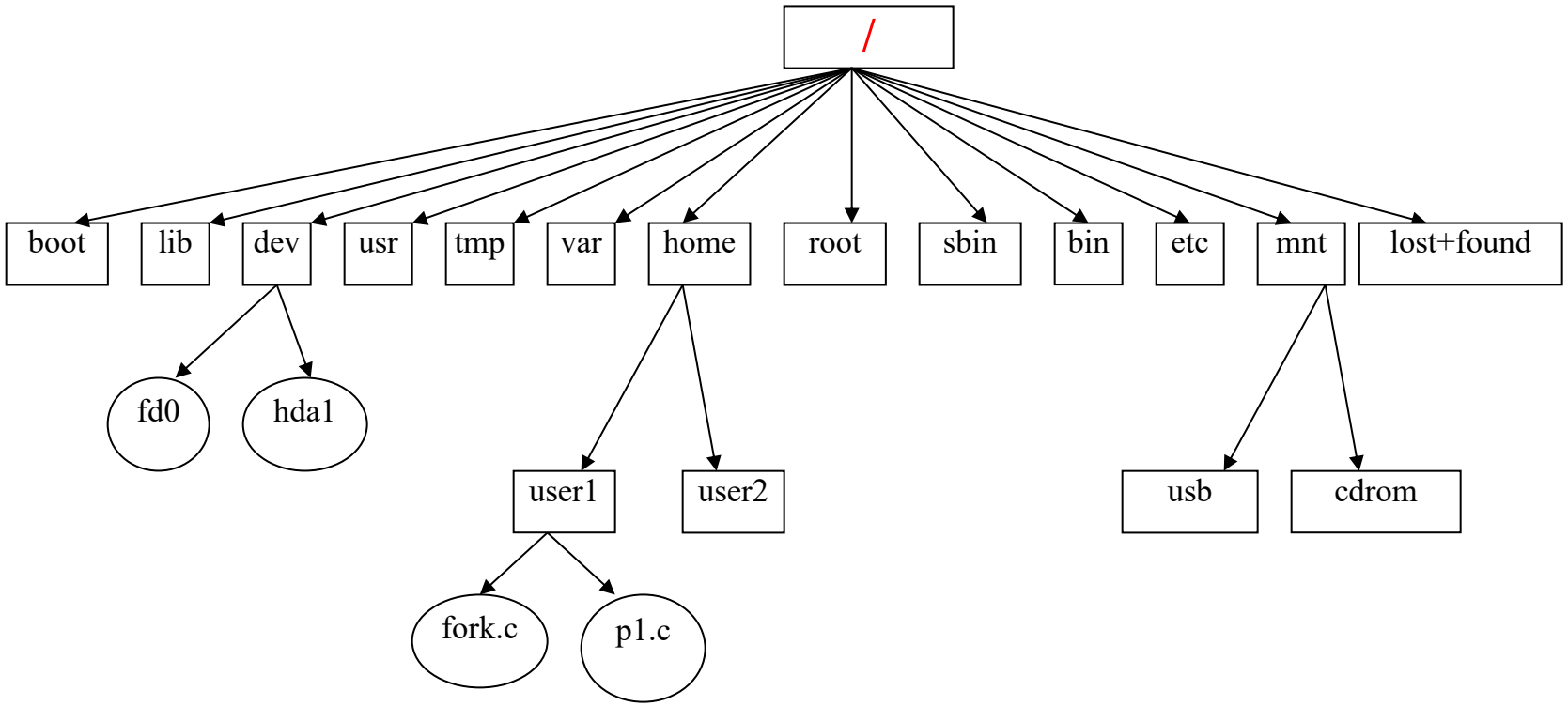


Linux文件系统概述

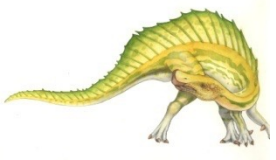
- 文件系统是对一个存储设备上的数据和元数据进行组织的机制。另一种看待文件系统的方式是把它看作一个协议
- Linux通过使用一组通用的 API 函数，可以在许多种存储设备上支持许多种文件系统。
- Linux文件系统采用了多级目录的树型层次结构管理文件。树型结构的最上层是根目录，用 / 表示。在根目录之下是各层目录和文件。
- Linux系统中的文件系统，不管是什么类型，都安装到一个目录下，并隐藏掉目录中原有的内容。这个目录叫做安装目录或者安装点。当文件系统卸载掉时，目录中的原有内容将再一次的显示出来。



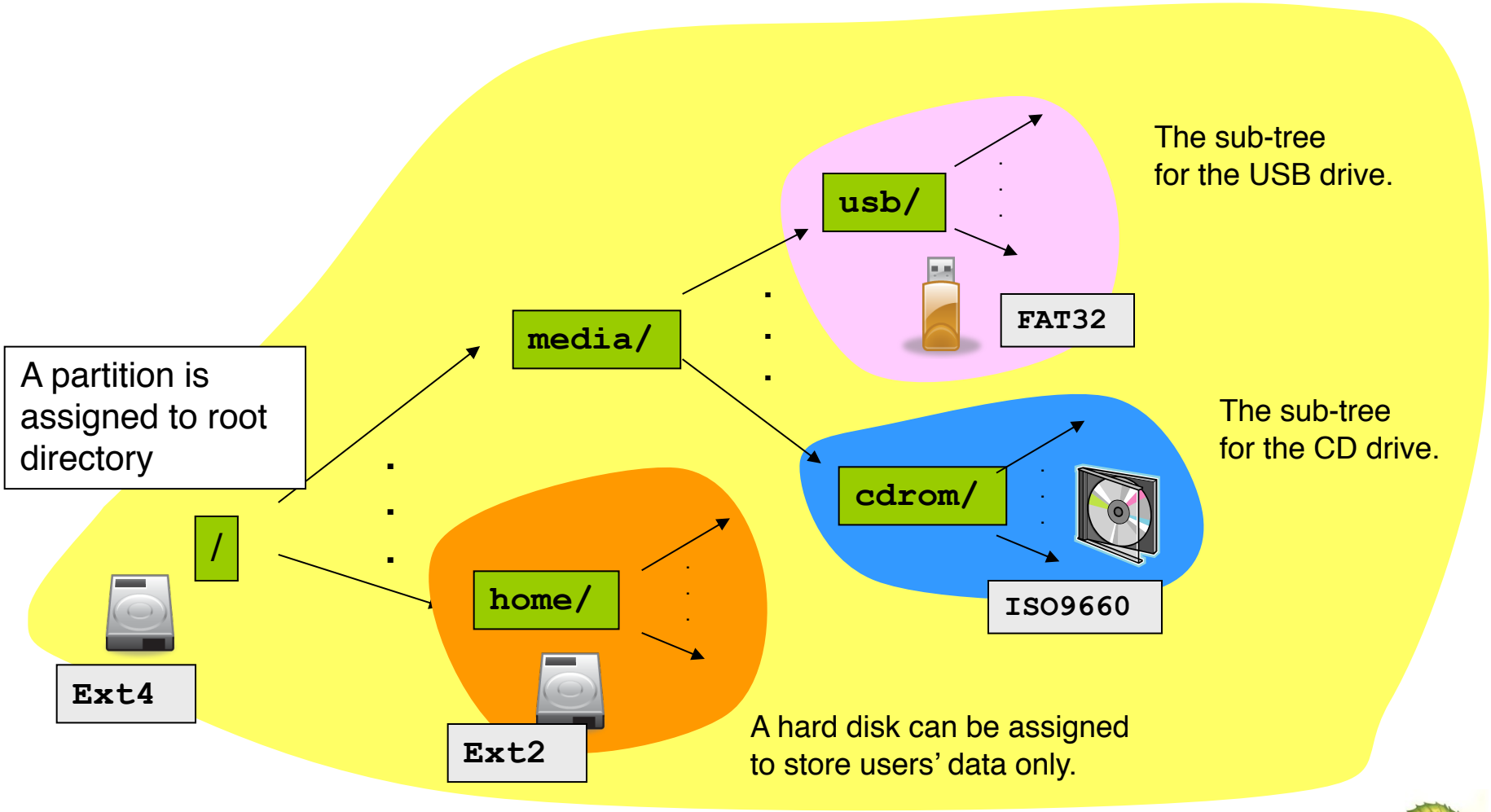
Linux文件系统目录结构

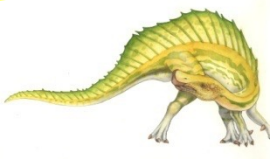


linux文件系统结构



File System and mount



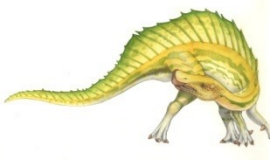


Linux文件系统概述

- 什么是文件、文件系统、目录、当前目录、绝对路径、相对路径。
- Linux缺省的文件系统（ext2、ext3、ext4）继承了UNIX，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，一个数据结构)。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。

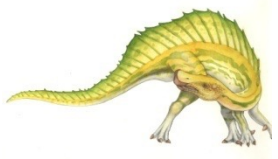
ext2目录项结构

```
/* Structure of a directory entry */
#define EXT2_NAME_LEN 255
/* The new version of the directory entry.  Since EXT2 structures are  stored in intel byte order, and the
        name_len field could never be  bigger than 255 chars, it's safe to reclaim the extra byte for the  file_type
        field. */
struct ext2_dir_entry_2 {
        __u32                   inode;                  /* Inode number */
        __u16                   rec_len;                /* Directory entry length */
        __u8            name_len;               /* Name length */
        __u8            file_type;
        char    name[EXT2_NAME_LEN];            /* File name */
};

/* Ext2 directory file types.  Only the low 3 bits are used.  The  other bits are reserved for now.*/
enum {
        EXT2_FT_UNKNOWN,
        EXT2_FT_REG_FILE,
        EXT2_FT_DIR,
        EXT2_FT_CHRDEV,
        EXT2_FT_BLKDEV,
        EXT2_FT_FIFO,
        EXT2_FT_SOCK,
        EXT2_FT_SYMLINK,
        EXT2_FT_MAX
};
```



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 by C 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)*

设备文件在哪个目录下？

- 管道(PIPE)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

文件的访问权限

- 3种用户和3种访问权限：

User Type	Permission Type		
	Read (r)	Write (w)	Execute (x)
User (u)	X	X	X
Group (g)	X	X	X
Others (o)	X	X	X

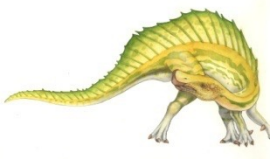
- 用 ls -l 或ls -ld命令显示文件的访问权限：

-rwxr--r-- 1 sarwar faculty 163 may 05 23:13 temp

文件类型 | 访问权限 | 链接计数 | 所有者 | 所在组 | 文件大小 | 日期 | 时间 | 文件名

文件系统类型

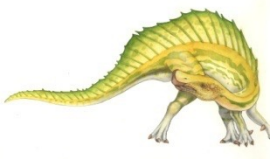
- 文件系统分三大类：
  - 基于磁盘的文件系统，如ext2/ext3/ext4、VFAT、NTFS等。
  - 网络文件系统，如NFS等。
  - 特殊文件系统，如proc文件系统、devfs、sysfs（/sys）等。
- 支持多种不同类型的文件系统是Linux操作系统的一大特色。如：ext、ext2、ext3、ext4、minix、iso9660 、 hpfs 、 msdos 、 vfat 、proc 、 nfs 、smb 、 sysv 、ntfs 、ufs、jfs、yaffs 、ReiserFS、CRAMFS、JFFS2等。
- Linux在标准内核中已支持的文件系统超过50种。
- 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。



/proc 文件系统

- /proc 虚拟文件系统，在这里可以获取系统状态信息并且修改系统的某些配置信息。
- 如内存情况在/proc/meminfo文件中，使用命令

  cat /proc/meminfo



查看Linux内核状况

系统信息

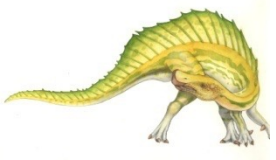
- procinfo命令显示大量的系统信息
- /proc下文件、目录的意义，在第12章proc文件系统中介绍
- /proc/sys目录是一个特殊目录，支持直接使用文件系统的操作，可以更改一些系统配置， 如：/proc/sys/fs/file-max

进程信息

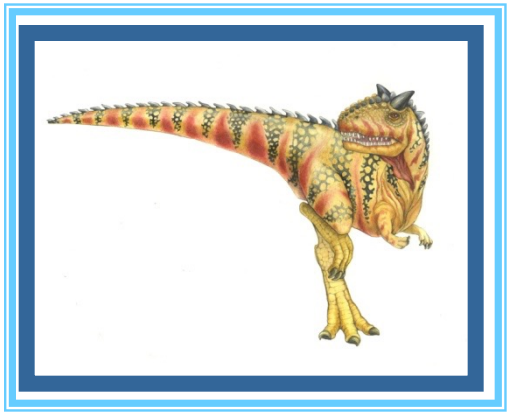
- /proc/self 是当前进程目录的符号链接。
- status文件包含PCB中的许多状态信息。用命令查看：

  cat /proc/self/ status

- /proc/<pid>/ 下文件、目录的意义，在第12章proc文件系统中介绍

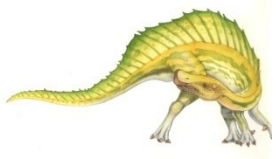


VFS虚拟文件系统



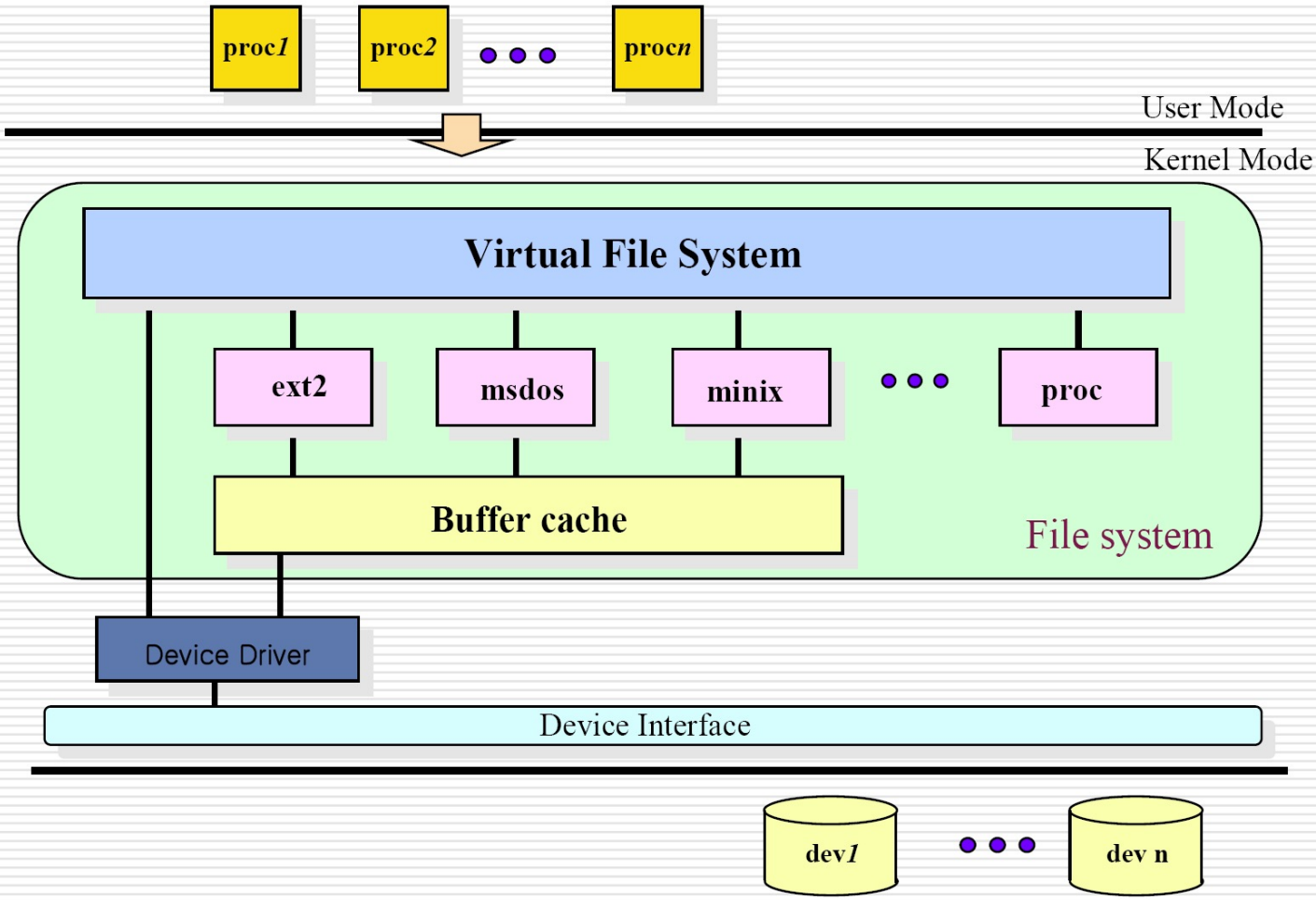
VFS虚拟文件系统

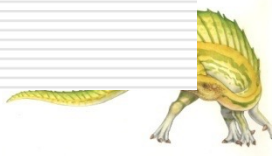
- Linux把各种不同的物理文件系统的所有特性进行抽象，建立起一个面向各种物理文件系统的转换机制，通过这个转换机制，把各种不同物理文件系统转换为一个具有统一共性的虚拟文件系统。这种转换机制称为虚拟文件系统转换VFS(Virtual Filesystem Switch/System) 。
- VFS并不是一种实际的文件系统。ext2/ext4等物理文件系统是存在于外存空间的，而VFS仅存在于内存。
- 在 VFS 上面，是对诸如 open、close、read 和 write 之类的函数的一个通用 API 抽象。在 VFS 下面是文件系统抽象，它定义了上层函数的实现方式。文件系统的源代码可以在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



虚拟文件系统VFS

Layers in the file system



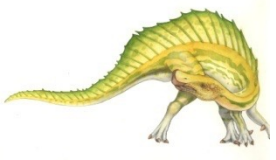


VFS的作用，例：

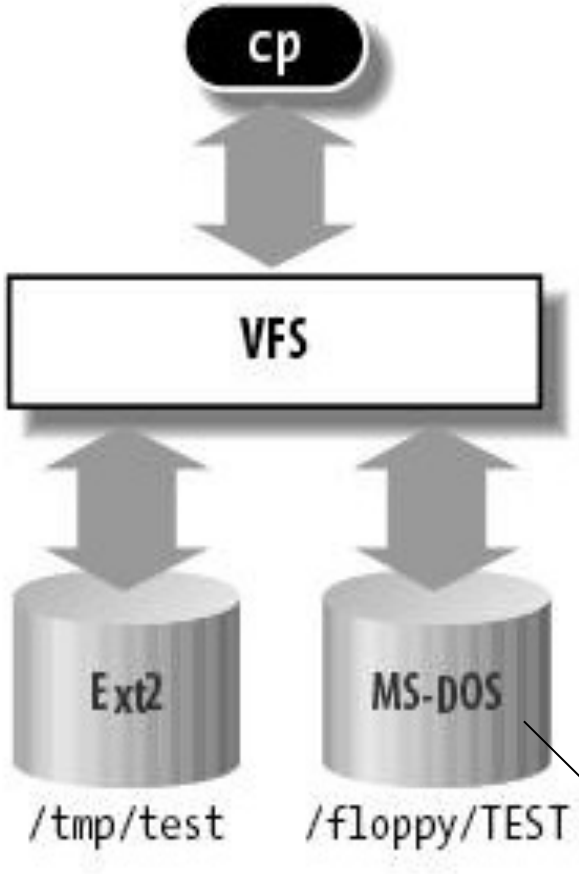
- 假设用户输入以下shell命令

  $ cp /floppy/TEST /tmp/test

  - 其中：
    - /floppy是MS-DOS的磁盘的一个挂载点（安装点）
    - /tmp是ext2文件系统中的一个目录
  - 对于cp命令而言，它不需要知道/floppy/TEST和/tmp/test分别是什么文件系统类型
  - cp程序通过VFS提供的系统调用接口进行文件操作



VFS role in a simple file copy operation



(a)

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

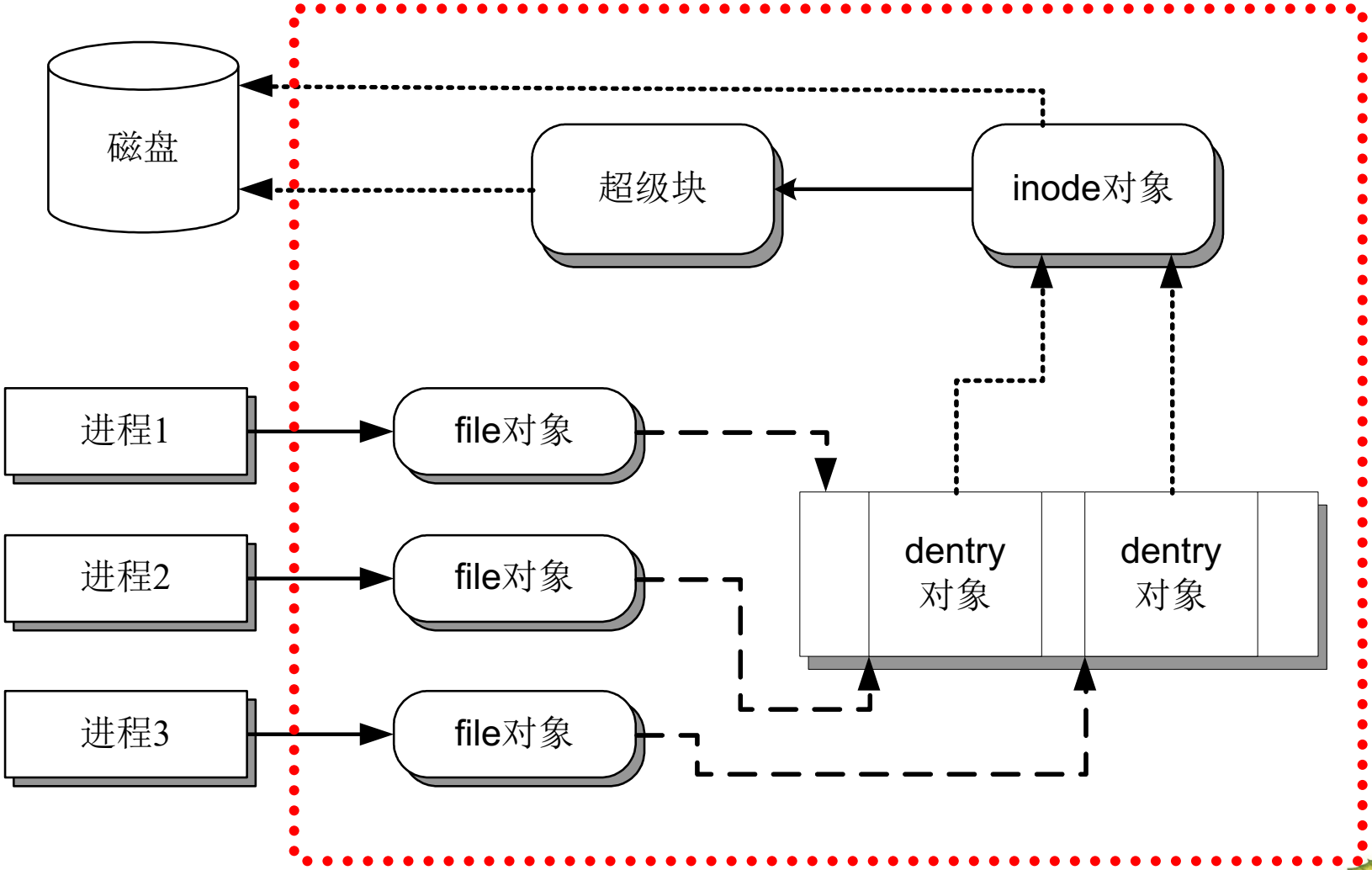
(b)

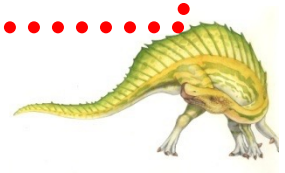
FAT文件系统

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。

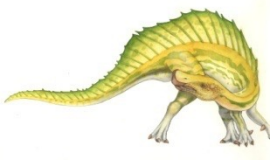
进程与VFS的交互


磁盘
超级块
inode对象
进程1
file对象
进程2
file对象
进程3
file对象
dentry
对象
dentry
对象



VFS的超级块

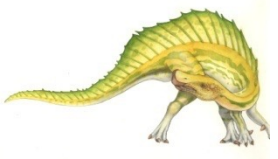
- 超级块superblock是文件系统中描述整体组织和结构的信息体。
- VFS把不同文件系统中的整体组织和结构信息，进行抽象后形成了兼顾不同文件系统的统一的超级块结构。
- VFS超级块是各种具体文件系统在安装时建立的，并在卸载时被自动删除。
- Linux中对于每种已安装的文件系统，在内存中都有与其对应的超级块。VFS超级块中的数据主要来自该文件系统的超级块。
- VFS超级块的数据结构定义是super_block结构。



super_block

include/linux/fs.h, line 805

```
805 struct super_block {
806         struct list_head        s_list;         /* Keep this first */
807         dev_t                   s_dev;          /* search index; _not_ kdev_t */
808         unsigned long           s_blocksize;
809         unsigned char           s_blocksize_bits;
810         unsigned char           s_dirt;
811         unsigned long long      s_maxbytes;     /* Max file size */
812         struct file_system_type *s_type;
813         struct super_operations *s_op;
...
818         unsigned long           s_magic;
819         struct dentry           *s_root;
...
829         struct list_head        s_inodes;       /* all inodes */
830         struct list_head        s_dirty;        /* dirty inodes */
831         struct list_head        s_io;           /* parked for writeback */
832         struct hlist_head       s_anon;         /* anonymous dentries for (nfs) exporting */
833         struct list_head        s_files;
...
844         void                    *s_fs_info;     /* Filesystem private info */
...
850         struct semaphore s_vfs_rename_sem;      /* Kludge */
...
```



super_block

s_list：指向了超级块链表中前一个超级块和后一个超级块的指针。

s_dev：超级块所在的设备的描述符。

s_blocksize和s_blocksize_bits：指定了磁盘文件系统的块的大小。

s_dirty：超级块的“脏”位。

s_maxbytes： 文件最大的大小。

s_type：指向文件系统的类型的指针。

s_op： 指向超级块操作的指针。

s_root：指向目录的dentry项。

s_dirt：表示“脏”（内容被修改了，但尚未被刷新到磁盘上）的inode节点的链表，分别指向前一个节点和后一个节点。

s_fs_info：指向各个文件系统私有数据，一般是各文件系统对应的超级块信息。以ext2文件系统为例，当ext2文件系统的超级块装入到内存，即装入到super_block的时候，会调用ext2_fill_super()函数，在这个函数中填写ext2对应的ext2_sb_info，然后挂在这个指针上。

VFS超级块的操作

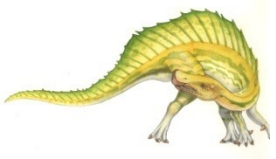
- 在系统运行中， VFS要建立、撤消一些VFS inode，还要对VFS超级块进行一些必要的操作。这些操作由一系列操作函数实现。
- 不同类型的文件系统的组织和结构不同，完成同样功能的操作函数的代码不同，每种文件系统都有自己的操作函数。
- 如何在对某文件系统进行操作时就能调用该文件系统的操作函数呢？这是由VFS接口通过转换实现的。
- 在VFS超级块中s_op是一个指向super_operations结构的指针，super_operations中包含着一系列的操作函数指针，即这些操作函数的入口地址。
- 每种文件系统VFS超级块指向的super_operations中记载的是该文件系统的操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

<include/linux/fs.h>

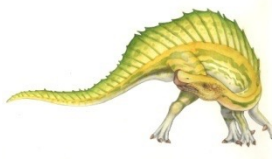
```
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);
        void (*read_inode) (struct inode *);
        void (*dirty_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*sync_fs)(struct super_block *sb, int wait);
        void (*write_super_lockfs) (struct super_block *);
        void (*unlockfs) (struct super_block *);
        int (*statfs) (struct super_block *, struct kstatfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
        int (*show_options)(struct seq_file *, struct vfsmount *);
};
```

read_inode is replaced by lookup->iget



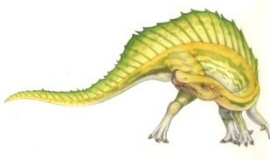
super_operations

- **read_inode()**：用磁盘上读取的信息来填充inode对象的内容，读取的inode结构中的i_ino对象可以用来在磁盘上定位对应的inode节点。
- **dirty_inode()**：表示一个inode对象已经“脏”。
- **write_inode()**：更新inode的信息，将其转换为磁盘相关的信息并写回。
- **put_inode()**：当有人释放inode对象引用的时候被调用，但是并不一定表示这个inode没人使用了，只是使用者减少了一个。
- **delete_inode()**：当inode的引用计数到达0的时候被调用，表明这个inode对应的对象可以被删除。删除磁盘的数据块，磁盘的inode以及VFS的inode。

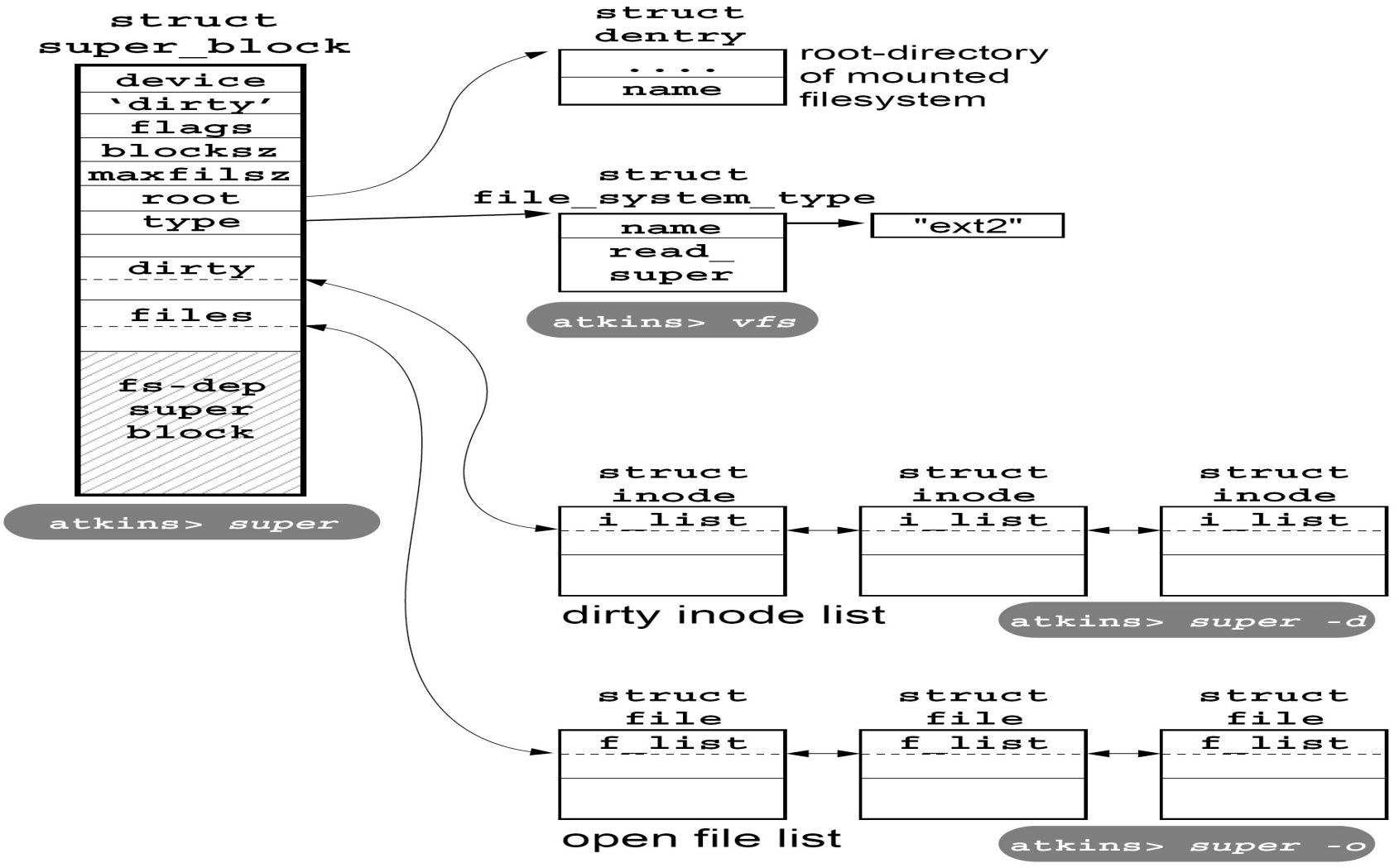


super_operations

- **put_super()**：由于当前的文件系统的卸载而释放当前的超级块对象。
- **write_super()**：更新当前的超级块对象的内容。
- **statfs()**：返回当前mount的文件系统的一些统计信息
- **remount_fs()**：按照一定的选项重新mount文件系统
- **clear_inode()**：和put_inode类似，但是也删除包含数据在内的内存对应inode中的结构。
- **umount_begin()**：开始umount操作，并中断其它的mount操作，用于网络文件系统。

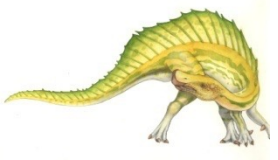


Incore superblock



VFS的inode对象

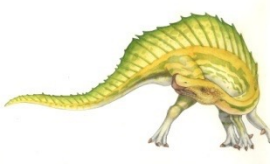
- Linux以ext2/ext3/ext4作为基本的文件系统，所以它的虚拟文件系统VFS中也设置了inode结构
- 物理文件系统的inode在外存中并且是长期存在的， VFS的inode 对象在内存中，它仅在需要时才建立，不再需要时撤消。
- 物理文件系统的inode是静态的，而VFS的inode是一种动态结构。
- inode结构定义在文件<include/linux/fs.h>中：



inode 对象

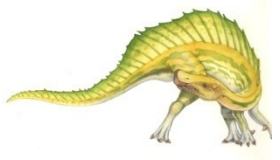
include/linux/fs.h

```
struct inode {
    struct list_head    i_hash; /* inode hash链表指针 */
    struct list_head    i_list;   /* inode链表指针 */
    struct list_head    i_dentry; /*dentry链表*/
     kdev_t             i_dev;   /* 主设备号*/
     unsigned long      i_ino; /* 外存的inode号 */
     umode_t            i_mode;   /* 文件类型和访问权限 */
     nlink_t            i_nlink;  /* 该文件的链接数 */
     uid_t              i_uid;    /* 文件所有者的用户标识 */
     gid_t              i_gid;    /* 文件的用户组标识 */
     kdev_t             i_rdev;   /* 次设备号 */
     off_t              i_size;   /* 文件长度，以字节为单位 */
     time_t             i_atime;  /* 文件最后一次访问时间 */
     time_t             i_mtime;  /* 文件最后一次修改时间 */
     time_t             i_ctime;  /* 文件创建时间 */
     unsigned long      i_blksize; /* 块尺寸，以字节为单位 */
     unsigned long      i_blocks;  /* 文件的块数 */
     unsigned long      i_version; /* 文件版本号 */
     unsigned long      i_nrpages; /* 文件在内存中占用的页面数 */
     struct semaphore                i_sem;  /* 文件同步操作用的信号量 */
     struct inode_operations         *i_op; /* 指向inode操作函数入口表的指针 */
     struct super_block              *i_sb;  /* 指向该文件系统的VFS超级块 */
     struct wait_queue               *i_wait;  /* 文件同步操作用等待队列 */
     struct file_lock    *i_flock;  /* 指向文件锁定链表的指针 */
```



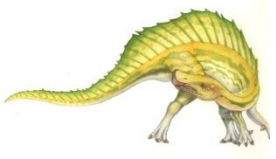
inode 对象

```
struct vm_area_struct *i_mmap; /* 文件使用的虚存区域 */
struct page *i_pages;   /* 指向文件占用内存页面page结构体链表 */
struct dquot *i_dquot[MAXQUOTAS];
struct inode *i_bound_to, *i_bound_by;
struct inode *i_mount;  /* 指向该文件系统根目录inode的指针 */
unsigned long i_count;          /* 使用该inode的进程计数 */
unsigned short i_flags;  /* 该文件系统的超级块标志 */
unsigned short i_writecount;/* 写计数 */
unsigned char i_lock;   /* 对该inode的锁定标志 */
unsigned char i_dirt;   /* 该inode的修改标志 */
unsigned char i_pipe;   /* 该inode表示管道文件 */
unsigned char i_sock;   /* 该inode表示套接字 */
unsigned char i_seek;   /* 未使用 */
unsigned char i_update;  /* inode更新标志 */
unsigned char i_condemned;
```



inode 对象

- VFS的inode与某个文件的对应关系是通过设备号i_dev与inode号i_ino建立的，它们唯一地指定了某个设备上的一个文件或目录。
- VFS的inode是物理设备上的文件或目录的inode在内存中的统一映像。这些特有信息是各种文件系统的inode在内存中的映像。如EXT2的ext2_inode_info结构。
- i_lock表示该inode被锁定，禁止对它的访问。i_flock表示该inode对应的文件被锁定。i_flock是个指向file_lock结构链表的指针，该链表指出了一系列被锁定的文件。
- VFS的inode组成一个双向链表，全局变量first_inode指向链表的表头。在这个链表中，空闲的inode总是从表头加入，而占用的inode总是从表尾加入。
- 系统还设置了一些管理inode 对象的全局变量，如：
  - max_inodes给定了inode的最大数量，
  - nr_inodes表示当前使用的inode数量，
  - nr_free_inodes表示空闲的inode数量。



inode 对象操作函数

- 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i_op指向inode_operations结构。
- 不同文件系统配备了自己的一整套inode操作函数。函数的入口地址记录在各自的inode_operations结构体中。

include/linux/fs.h, line 1029

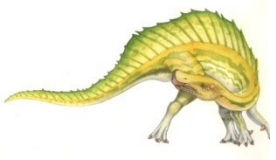
```
1029 struct inode_operations {
1030        int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
1031        struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
1032        int (*link) (struct dentry *,struct inode *,struct dentry *);
1033        int (*unlink) (struct inode *,struct dentry *);
1034        int (*symlink) (struct inode *,struct dentry *,const char *);
1035        int (*mkdir) (struct inode *,struct dentry *,int);
1036        int (*rmdir) (struct inode *,struct dentry *);
1037        int (*mknod) (struct inode *,struct dentry *,int,dev_t);
1038        int (*rename) (struct inode *, struct dentry *,
1039                       struct inode *, struct dentry *);
1040        int (*readlink) (struct dentry *, char __user *,int);
```

inode_operations

```
1041  void * (*follow_link) (struct dentry *, struct nameidata *);
1042  void (*put_link) (struct dentry *, struct nameidata *, void *);
1043  void (*truncate) (struct inode *);
1044  int (*permission) (struct inode *, int, struct nameidata *);
1045  int (*setattr) (struct dentry *, struct iattr *);
1046  int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
1047  int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
1048  ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
1049  ssize_t (*listxattr) (struct dentry *, char *, size_t);
1050  int (*removexattr) (struct dentry *, const char *);
1051  void (*truncate_range)(struct inode *, loff_t, loff_t);
1052 };
```

inode_operations

- 针对目录inode操作函数：
  - create：只适用于目录inode，当VFS需要在“inode”里面创建一个文件（文件名在dentry里面给出）的时候被调用。VFS必须已经检查过文件名在这个目录里面不存在。
  - lookup：用于检查一个文件（文件名在dentry里面给出）是否在一个inode目录里面。
  - link：在inode所给出的目录里面创建一个从第一个参数dentry文件到第三个参数dentry文件的硬链接（hard link）。
  - unlink：从inode目录里面删除dentry所代表的文件。
  - symlink：用于在inode目录里面创建软链接（soft link）。
  - mkdir：用于在inode目录里面创建子目录。
  - rmdir：用于在inode目录里面删除子目录。
  - mknod：用于在inode目录里面创建设备文件。

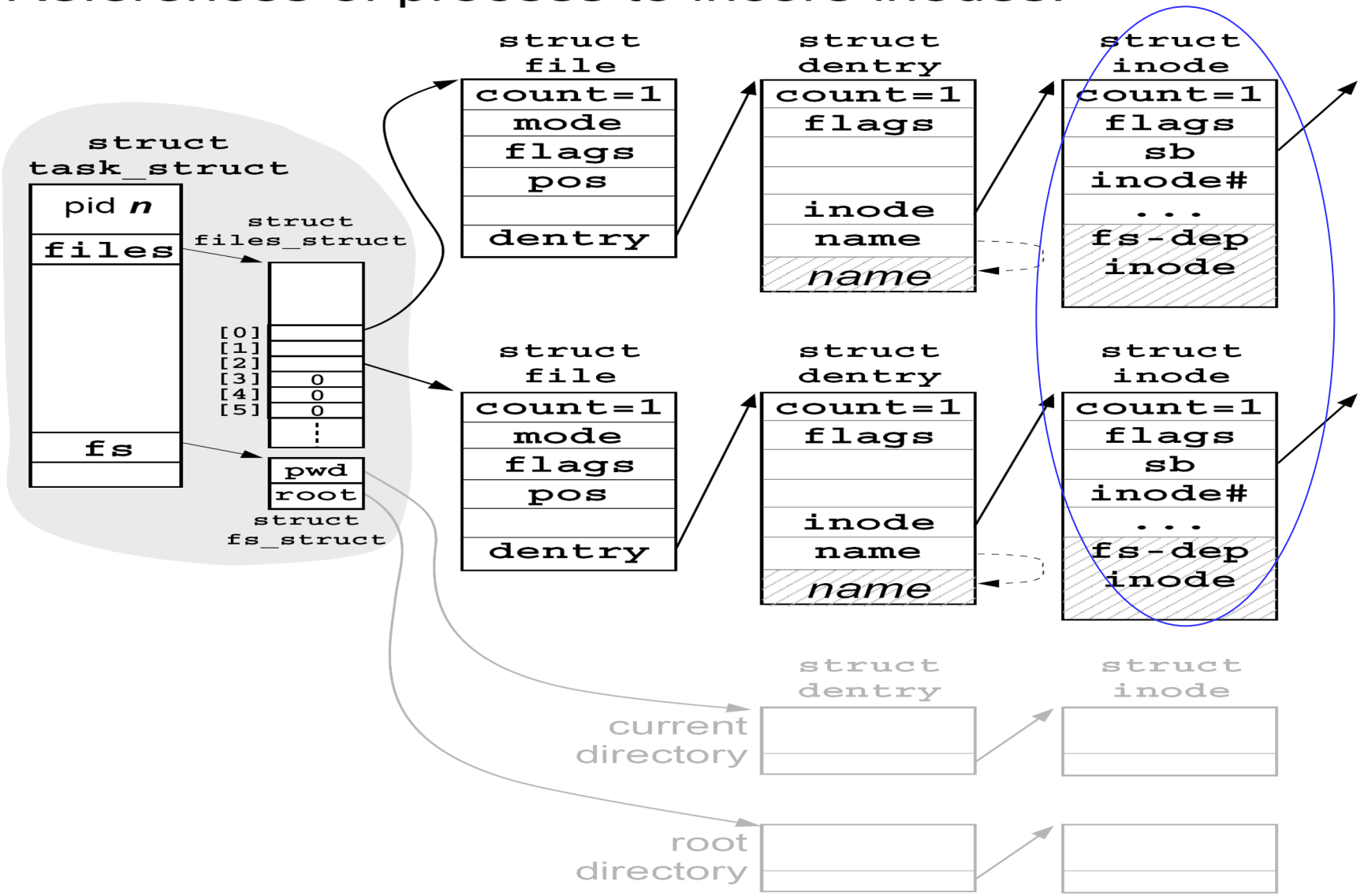


inode_operations

- 其他操作函数
  - rename：把第一个和第二个参数（inode，dentry）所定位的文件改名为第三个和第四个参数所定位的文件。
  - readlink：读取一个软链接所指向的文件名。
  - follow_link：VFS调用这个函数跟踪一个软链接到它所指向的inode。
  - put_link：VFS调用这个函数释放follow_link分配的一些资源。
  - truncate：VFS调用这个函数改变一个文件的大小。
  - permission：VFS调用这个函数得到对一个文件的访问权限。
  - setattr：VFS调用这个函数设置一个文件的属性。比如chmod系统调用就是调用这个函数。
  - getattr：查看一个文件的属性。比如stat系统调用就是调用这个函数。
  - setxattr：设置一个文件的某项特殊属性。详细情况请查看setxattr系统调用帮助。
  - getxattr：查看一个文件的某项特殊属性。详细情况请查看getxattr系统调用帮助。
  - listxattr：查看一个文件的所有特殊属性。详细情况请查看listxattr系统调用帮助。
  - removexattr：删除一个文件的特殊属性。详细情况请查看removexattr系统调用帮助。

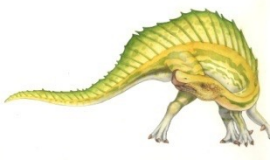
Processes and incore inodes

References of process to incore inodes:



目录项对象dentry object

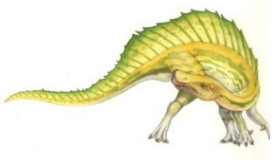
- 每个文件除了有一个索引节点inode数据结构外，还有一个目录项dentry数据结构。
- 每个dentry代表路径中的一个特定部分。如：/、bin、vi都属于目录项对象。
- 目录项也可包括安装点，如：/mnt/cdrom/foo，/、mnt、cdrom、foo都属于目录项对象。
- 目录项对象作用是帮助实现文件的快速定位，还起到缓冲作用

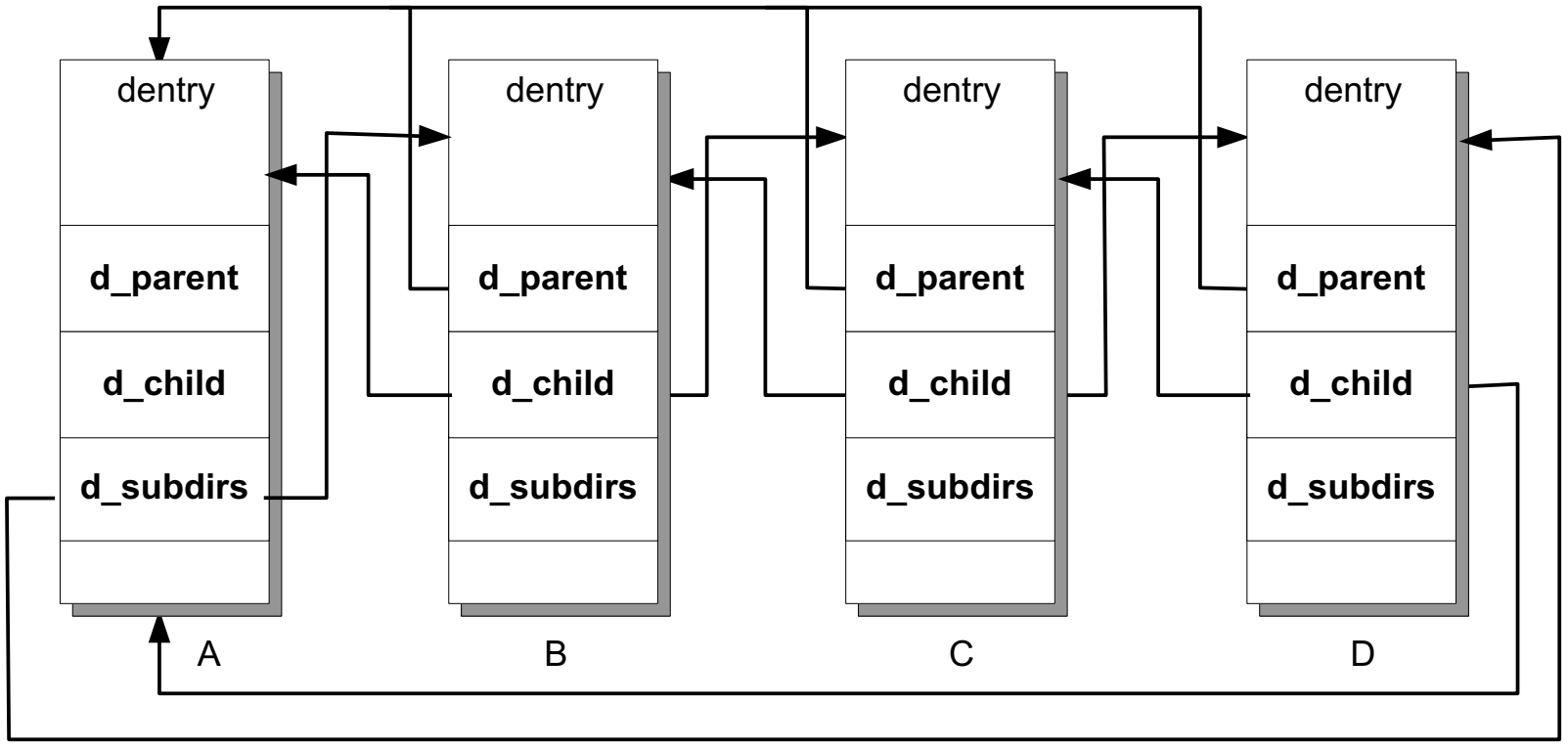


目录项对象

- <include/linux/dcache.h>

```
struct dentry {
  atomic_t d_count;       /* 目录项引用计数器 */
  unsigned int d_flags;   /* 目录项标志 */
  struct inode  * d_inode;  /* 与文件名关联的索引节点 */
  struct dentry * d_parent;     /* 父目录的目录项 */
  struct list_head d_hash;      /* 目录项形成的哈希表 */
  struct list_head d_lru;       /*未使用的 LRU 链表 */
  struct list_head d_child;     /*父目录的子目录项所形成的链表 */
  struct list_head d_subdirs;   /* 该目录项的子目录所形成的链表*/
  struct list_head d_alias;     /* 索引节点别名的链表*/
  int d_mounted;                /* 目录项的安装点 */
  struct qstr d_name;           /* 目录项名（可快速查找） */
  struct dentry_operations  *d_op;/* 操作目录项的函数*/
  struct super_block * d_sb;    /* 目录项树的根（即文件的超级块）*/
  unsigned long d_vfs_flags;
  void * d_fsdata;              /* 具体文件系统的数据 */
unsigned char d_iname[DNAME_INLINE_LEN]; /* 短文件名 */
……
};
```





dentry的结构和指针（其中**A**为父节点目录，**B, C, D**为它的三个儿子）

目录项对象

- 对目录项进行操作的一组函数，由d_op指向dentry_operation结构：

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, int);
        int (*d_hash) (struct dentry *, struct qstr *);
        int (*d_compare) (struct dentry *, struct qstr *,
    struct qstr *);
        int (*d_delete)(struct dentry *);
        void (*d_release)(struct dentry *);
        void (*d_iput)(struct dentry *, struct inode *);
};
```

目录项对象

- 该结构中函数的主要功能简述如下：

  d_revalidate（）：判定目录项是否有效。

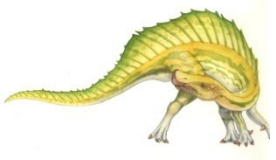
  d_hash（）：生成一个哈希值。

  d_compare（）：比较两个文件名

  d_delete（）：删除d_count域为0 的目录项对象

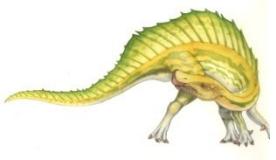
  d_release（）释放一个目录项对象。

  d_iput（）： 调用该方法丢弃目录项对应的索引节点



VFS的dentry cache 与 inode cache

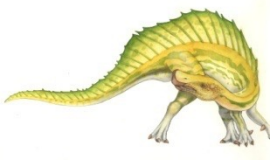
- 为了加速对经常使用的目录的访问， VFS文件系统维护着一个目录项的缓存。
- 为了加快文件的查找速度VFS文件系统维护一个inode节点的缓存以加速对所有装配的文件系统的访问。
- 用hash表将缓存对象组织起来。



file对象

- 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。
- file结构定义在文件<include/linux/fs.h>中：

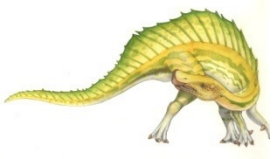
```
struct file {
 struct list_head f_list;  /*file结构链表*/
 struct dentry   *f_dentry;/*指向与文件对象关联的dentry对象*/
 struct vfsmount *f_vfsmnt; /*文件相应的vfsmount结构*/
 struct file_operations *f_op; /*文件对象的操作集合*/
 atomic_t f_count ; /*文件打开的引用计数*/
 unsigned int f_flags; /*使用open（）时设定的标志*/
 mode_t  f_mode;  /*文件读写权限*/
 loff_t  f_pos;   /*对文件读写操作的当前位置*/
 struct fown_struct      f_owner;
    ......
};
```



file_operations

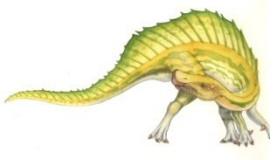
- include/linux/fs.h, line 999

```
999 struct file_operations {
1000        struct module *owner;
1001        loff_t (*llseek) (struct file *, loff_t, int);
1002        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
1003        ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
1004        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
1005        ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t,
        loff_t);
1006        int (*readdir) (struct file *, void *, filldir_t);
1007        unsigned int (*poll) (struct file *, struct poll_table_struct *);
1008    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
1009    ......
}
```



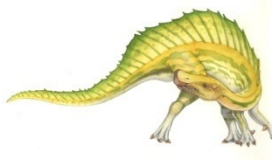
file_operations

- llseek：用于移动文件内部偏移量。
- read：读文件。
- aio_read：异步读，被io_submit和其他的异步IO函数调用。
- write：写文件。
- aio_write：异步写，被io_submit和其他的异步IO函数调用。
- readdir：当VFS需要读目录内容的时候调用这个函数。
- poll：当一个进程想检查一个文件是否有内容可读写的时候，VFS调用这个函数；一般来说，调用这个函数之后进程进入睡眠，直到文件中有内容读写就绪时被唤醒。详情请参考select和poll系统调用。
- ioctl：被系统调用ioctl调用。
- unlocked_ioctl：被系统调用ioctl调用；不需要BKL（内核锁）的文件系统应该使用这个函数，而不是上面那个ioctl。



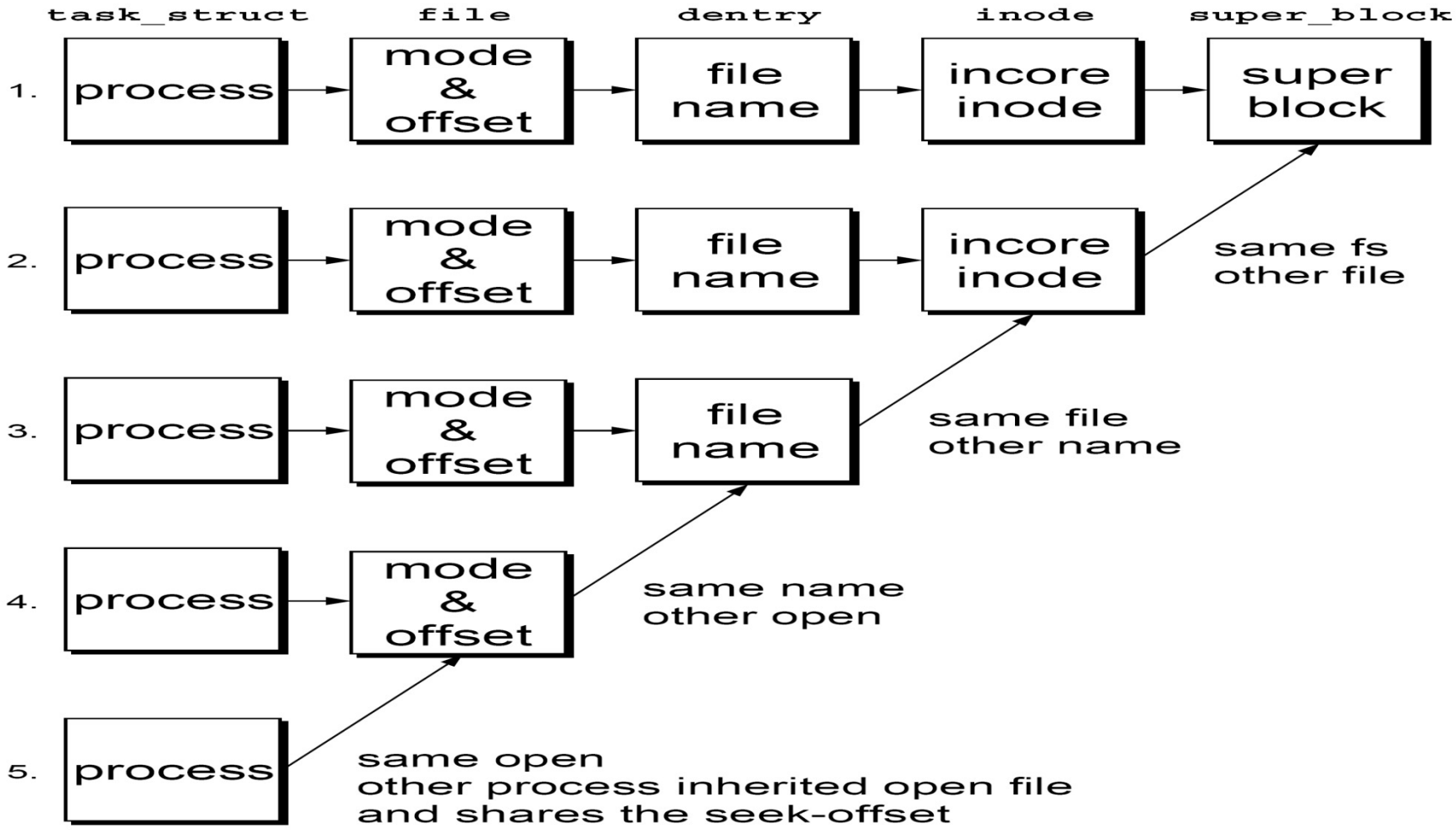
file_operations

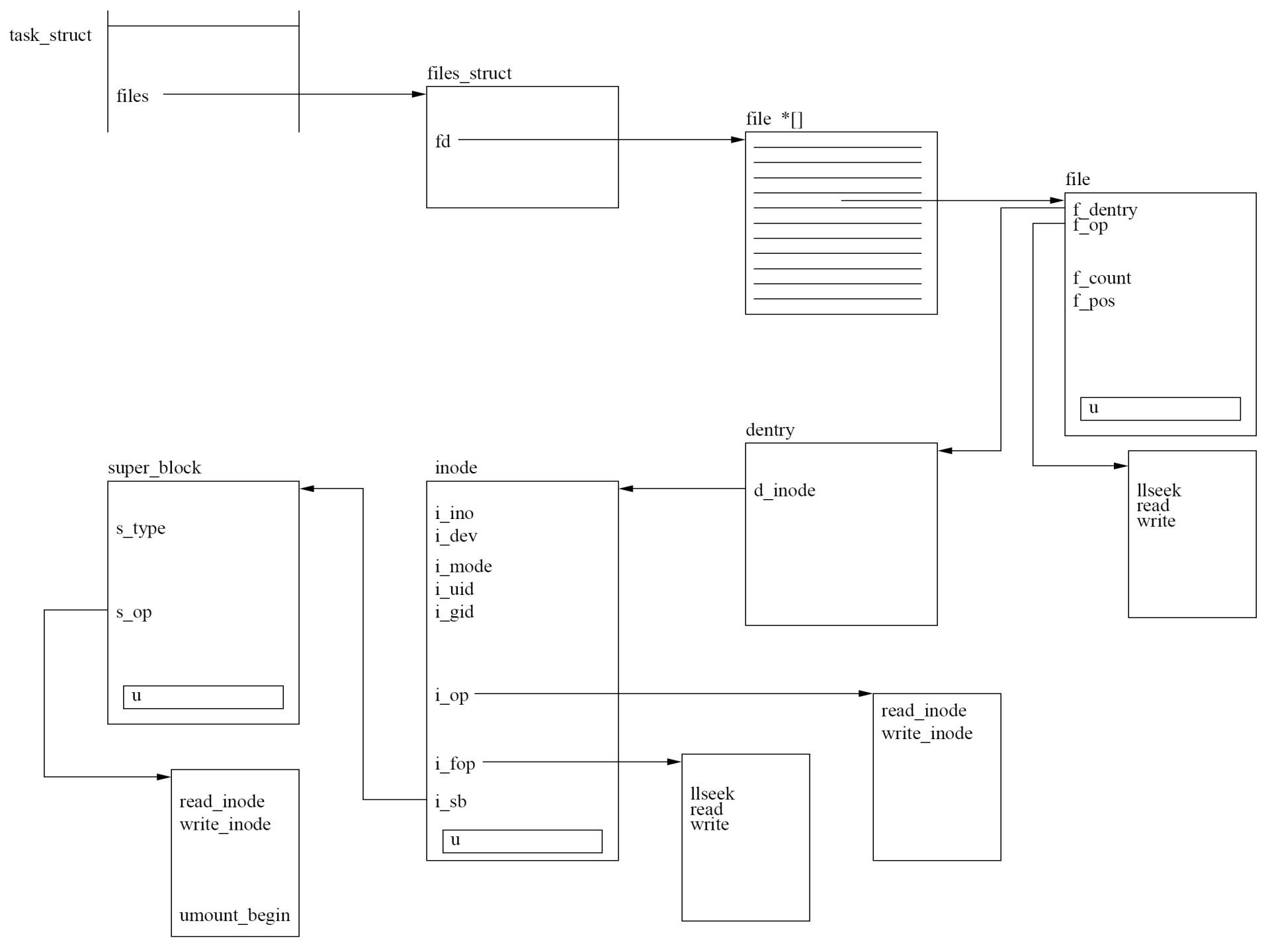
- compat_ioctl：被系统调用ioctl调用；当在64位内核上使用32位系统调用的时候使用这个ioctl函数。
- mmap：被系统调用mmap调用。
- open：通过创建一个新的文件对象而打开一个文件，并把它链接到相应的索引节点对象。
- flush：被系统调用close调用，把一个文件内容写回磁盘。
- release：当对一个打开文件的最后引用关闭的时候，VFS调用这个函数释放文件。
- fsync：被系统调用fsync调用。
- fasync：当对一个文件启用异步读写（非阻塞读写）的时候，被系统调用fcntl调用。
- lock：fcntl系统调用使用命令F_GETLK，F_SETLK和F_SETLKW的时候，调用这个函数。



Processes and open files

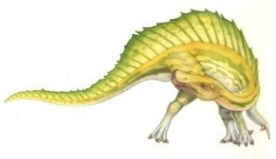
Overview entities in memory:




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

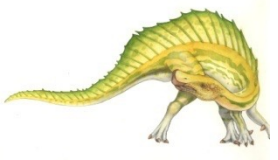
文件系统的注册

- Linux 支持的文件系统必须注册后才能使用，文件系统不再使用时则予以注销。
- 向系统内核注册有两种方式：
  - 一种是在系统引导时在VFS中注册，在系统关闭时注销。
  - 另一种是把文件系统做为可装卸模块，在安装时在VFS中注册，并在模块卸载时注销。
- 文件系统的注册由VFS中的注册链表进行管理。
- 每种注册的文件系统登记在file_system_type结构体中，file_system_type结构体组成一个链表，称为注册链表，链表的表头由全局变量file_system给出。



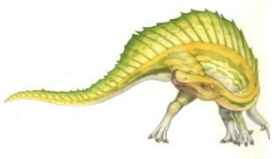
file_system_type结构

```
struct file_system_type {
    const char *name;
    int fs_flags;
    struct super_block *(*get_sb) (struct file_system_type *, int,
                                   const char *, void *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;
};
```



file_system_type结构

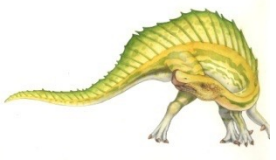
- The fields of the file_system_type object
  - name： 文件系统类型名字，比如“ext2”，“vfat”等等。
  - fs_flags：mount的文件系统的参数。
  - get_sb：当这种类型的文件系统要被mount的时候，这个函数会被调用，用以得到相应文件系统的超级块。
  - kill_sb：当这种类型的文件系统被umount的时候，这个函数被调用。
  - owner： VFS内部使用，大多数情况下，你只需要初始化为THIS_MODULE。
  - next： 文件系统类型链表的后续指针。VFS内部使用，初始化为NULL。
  - list_head fs_supers：文件系统的超级块的双向链表。



file_system_type结构

- 对于EXT2文件系统：

```
static struct file_system_type ext2_fs_type = {
    .owner          = THIS_MODULE,
    .name           = "ext2",
    .get_sb         = ext2_get_sb,
    .kill_sb        = kill_block_super,
    .fs_flags       = FS_REQUIRES_DEV,
};
```



文件系统的注册

- 文件系统的注册是通过内核提供的文件系统初始化函数实现的：

  init_ext2_fs()　ext2文件系统初始化函数；

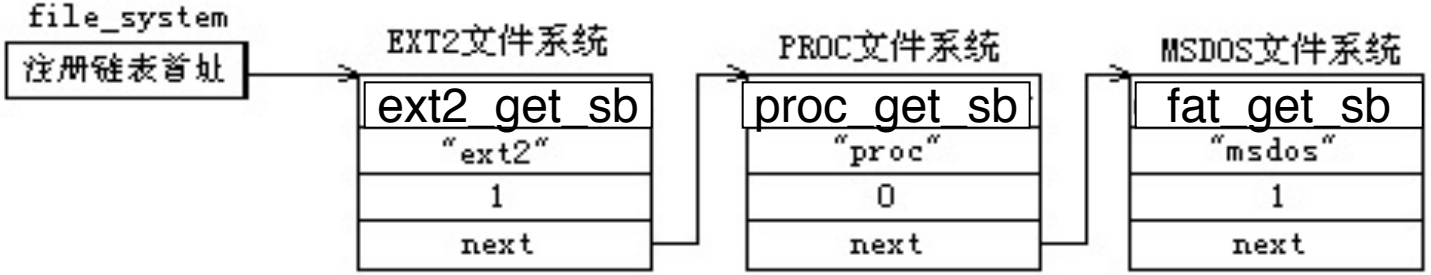
  init_minix_fs()　minix文件系统初始化函数；

  init_msdos_fs()　msdos文件系统初始化函数；

  init_proc_fs()　proc文件系统初始化函数；

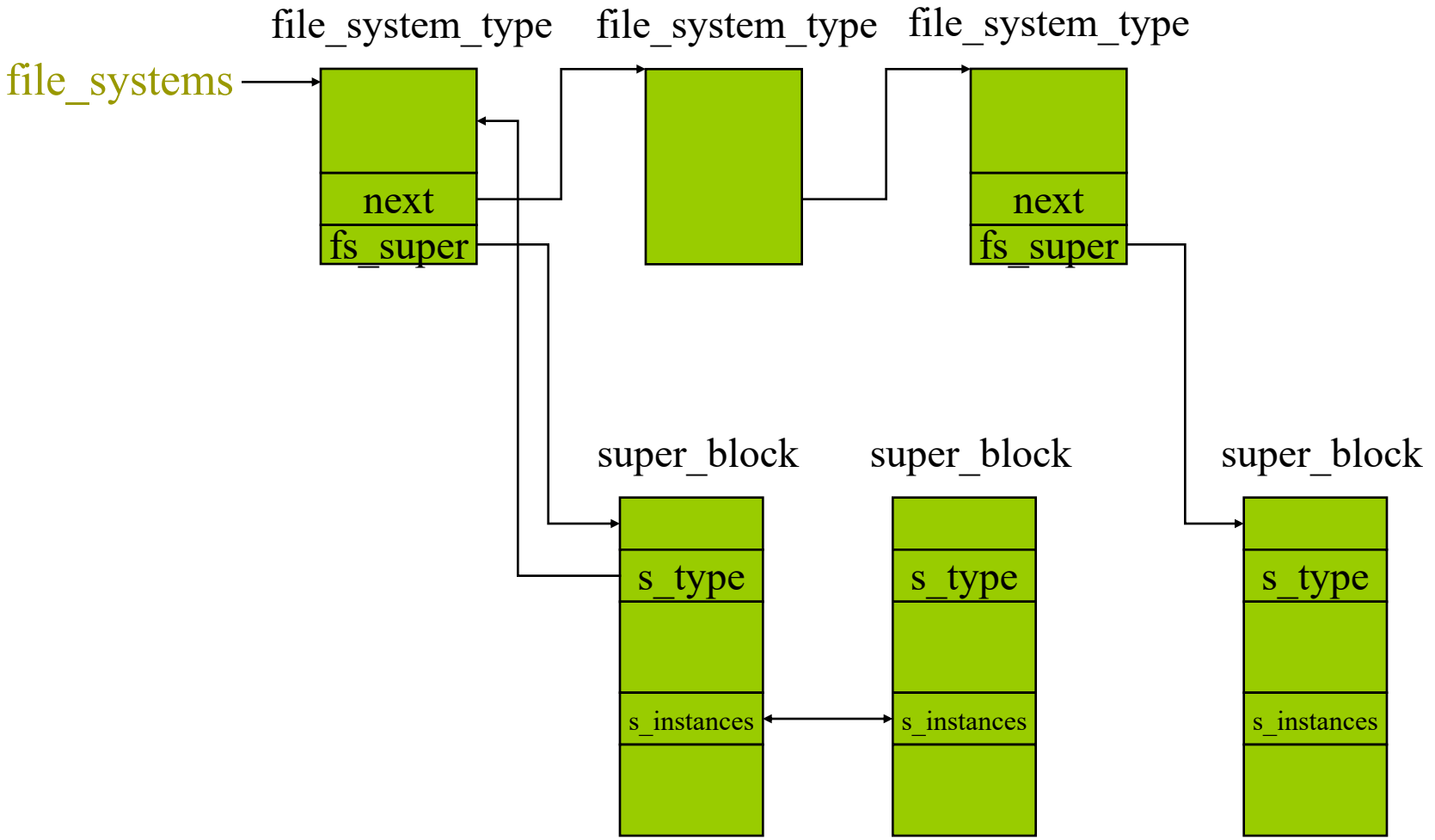
  init_sysv_fs()　sysv文件系统初始化函数；

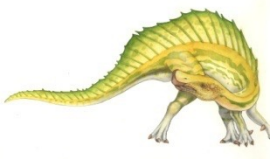
- 在文件系统初始化函数中，把注册结构体做为参数，　调用由内核提供的注册函数**register_filesystem**（）



Linux的注册链表

文件系统的注册


file_system_type
file_system_type
file_system_type
file_systems
next
fs_super
next
fs_super
super_block
super_block
super_block
s_type
s_type
s_type
s_instances
s_instances
s_instances



文件系统的安装

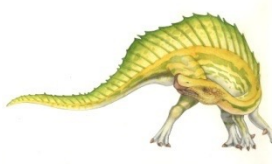
- 要使用磁盘文件系统必须安装到系统中。要安装的文件系统必须已经存在于外存磁盘上，每个文件系统占用一个独立的磁盘分区，并且具有各自的树型层次结构。
- 由于ext2/ext3/ext4是Linux的标准文件系统，所以系统把ext2/ext3/ext4文件系统的磁盘分区做为系统的根文件系统。其他文件系统则安装在根文件系统下的某个目录下，成为系统树型结构中的一个分支。
- Linux文件系统的树型层次结构中用于安装其它文件系统的目录称为安装点或安装目录。
- 文件系统的安装命令：

```
# mount –t vfat /dev/hda5 /mnt/win
```

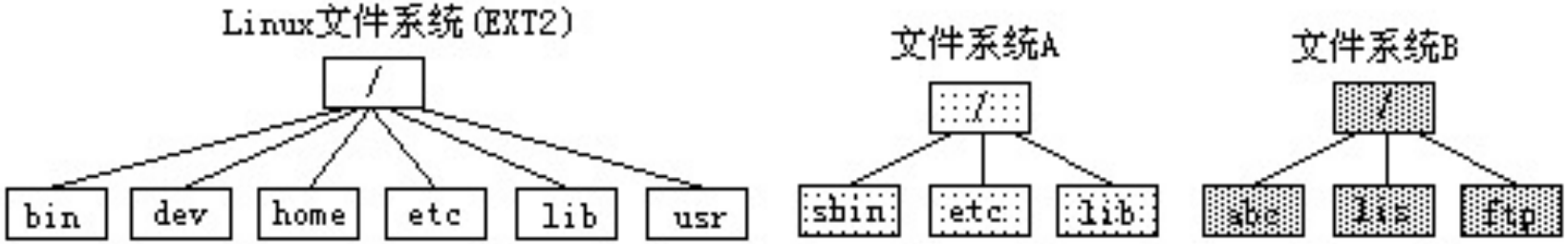
内核与之对应的系统调用是sys_mount()。该函数根据文件系统类型得到相应的file_system_type 对象,再取得该分区的超级块对象。

mount命令

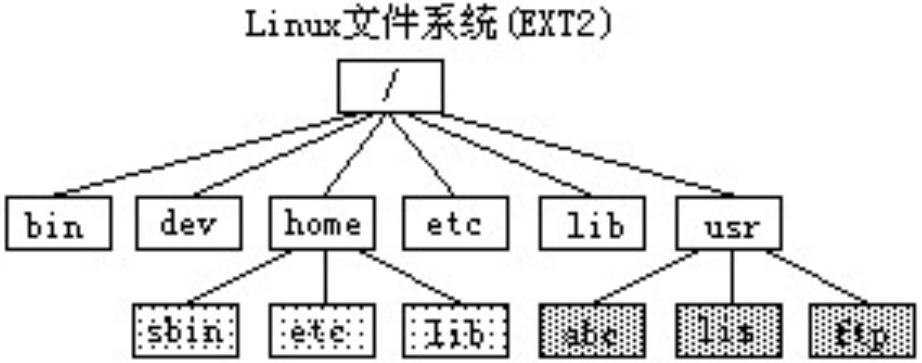
- mount：挂载文件系统
- 命令语法：
  - mount [-t fstype] [-o options] device dirname
- 常用参数：
  - fstype：文件系统类型，如：
    - iso9660 cd-rom使用的标准文件系统
    - vfat Windows操作系统的fat32文件系统
    - ntfs Windows的NTFS文件系统
  - device：设备文件，格式：/dev/xxyN
  - dirname：挂载目录
  - options：设备或文件的挂接方式。常用的参数有：
    - loop：把一个文件当成块设备挂接（环回设备）
    - ro：采用只读方式挂接设备
    - rw：采用读写方式挂接设备
    - iocharset：指定访问文件系统所用字符集
- 例：mount –t ext2 –o loop ./myfs /mnt



文件系统的安装



(a) 安装前的三个独立的文件系统



(b) 安装后的文件系统

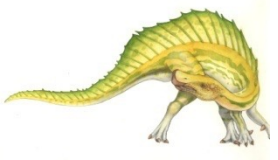
文件系统的安装

文件系统的安装

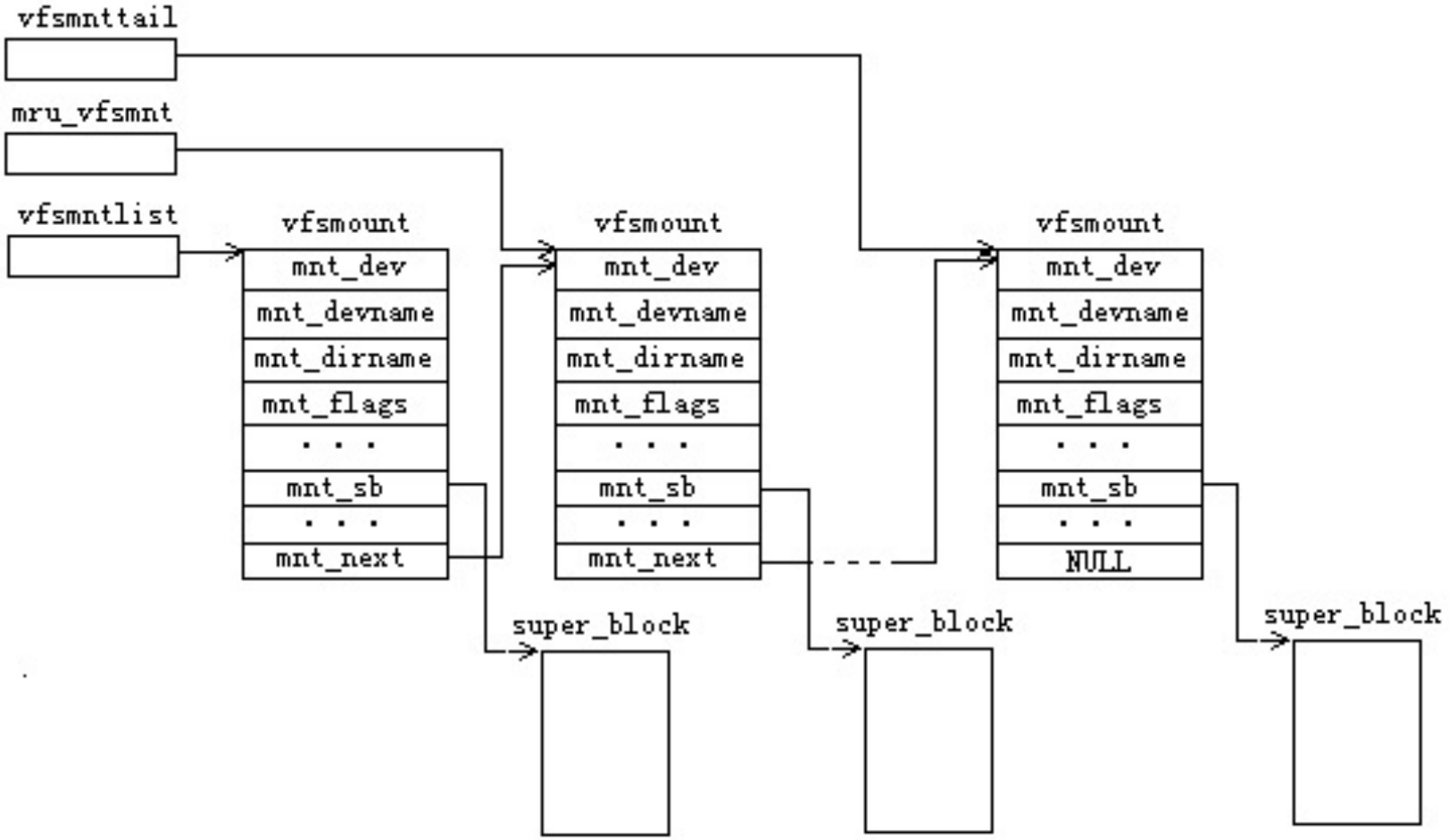
- 已安装的文件系统用一个**vfsmount**结构 **<include/linux/mount.h>**进行描述

vfsmount结构

```
struct vfsmount {
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent;            /* fs we are mounted on */
        struct dentry *mnt_mountpoint;          /* dentry of mountpoint */
        struct dentry *mnt_root;                /* root of the mounted tree */
        struct super_block *mnt_sb;             /* pointer to superblock */
        struct list_head mnt_mounts;            /* list of children, anchored here */
        struct list_head mnt_child;             /* and going through their mnt_child */
        atomic_t mnt_count;
        int mnt_flags;
        int mnt_expiry_mark;                    /* true if marked for expiry */
        char *mnt_devname;                      /* Name of device e.g. /dev/dsk/hda1 */
        struct list_head mnt_list;
        struct list_head mnt_expire;            /* link in fs-specific expiry list */
        struct list_head mnt_share;             /* circular list of shared mounts */
        struct list_head mnt_slave_list;        /* list of slave mounts */
        struct list_head mnt_slave;             /* slave list entry */
        struct vfsmount *mnt_master;            /* slave is on master->mnt_slave_list */
        struct namespace *mnt_namespace; /* containing namespace */
        int mnt_pinned;
};
```



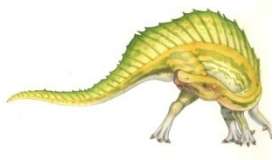
文件系统的安装



已安装文件系统的管理结构示意图

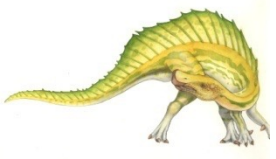
一些操作

- 已知inode号，读inode信息：
  - 因为每组的inode数目固定,所以很容易计算出该文件属于那个组并且得到在组中inode表的下标,继而可得到ext2_inode信息。
- 查找文件：
  - 以/root/temp.c为例。根目录的inode号总为2,因此可以得到根目录的ext2_inode信息,再从i_block[]指向的数据块查找是否有ext2_dir_entry_2项的名字等于root,如果有,则同时得到了root的inode号。重复上述过程,则可以判定是否有/root/temp.c存在。
- 读取文件某个位置的数据：
  - 给定的位置是相对于文件而不是相对于磁盘的,因此需要根据该位置计算出它在i_block中的下标, 得到在磁盘上的位置。



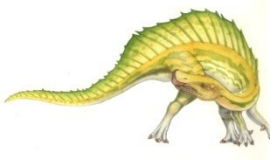
磁盘块的分配与释放

- **磁盘块的释放：**
  - 主要工作是修改块位图和涉及块的统计变量。
- **磁盘块的分配：**
  - 先试图分配与上次分配给文件的块相邻的块。
  - 如果不行,则试图在附近64个块的范围内分配。
  - 还不行的话，在本组内向前找八个连续空闲的块。
  - 若还不满足,则找任何空闲的块均可以。
  - 再不行,则到其他的组中去寻找。
- **目的：** 文件尽可能地连续分配使文件访问时间变短。



文件管理和操作

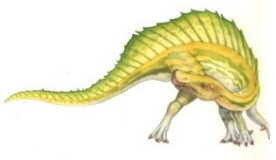
- 对于系统中打开的文件，主要从两个方面进行管理，一是由系统通过系统打开文件表进行统一管理，另一是由进程通过私有数据结构进行管理。文件打开后要进行各种操作，VFS提供了面向文件操作的统一接口。
- 系统打开文件表：
  - Linux系统内核把所有进程打开的文件集中管理， 把它们组成“系统打开文件表”。
  - 系统打开文件表是一个双向链表，它的每个表项（节点）是一个file结构。
  - 进程打开一个文件就建立一个file结构，并把它加入到系统打开文件链表中。
  - 全局变量first_file指向系统打开文件表的表头。



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结构记录着文件系统根目录和当前目录，
    - files_struct结构包含着进程的打开文件表。

```
struct fs_struct {
    int count;     /* 共享此结构的计数值 */
    unsigned short umask;  /* 掩码 */
    struct inode * root, * pwd;
        /* 根目录和当前目录inode指针 */
};
```



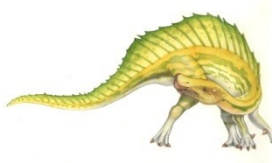
文件管理和操作

include/linux/fdtable.h

```
#define NR_OPEN 256
struct files_struct {
    int count;    /* 共享该结构的计数值 */
    fd_set close_on_exec;
    fd_set open_fds;
    struct file * fd[NR_OPEN];
};
```

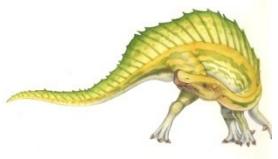
新内核fd_array

- fd[]每个元素是一个指向file结构体的指针，该数组称为进程打开文件表。
- 进程打开一个文件时，建立一个file结构体，并加入到系统打开文件表中，然后把该file结构体的首地址写入fd[]数组的第一个空闲元素中，一个进程所有打开的文件都记载在fd[]数组中。



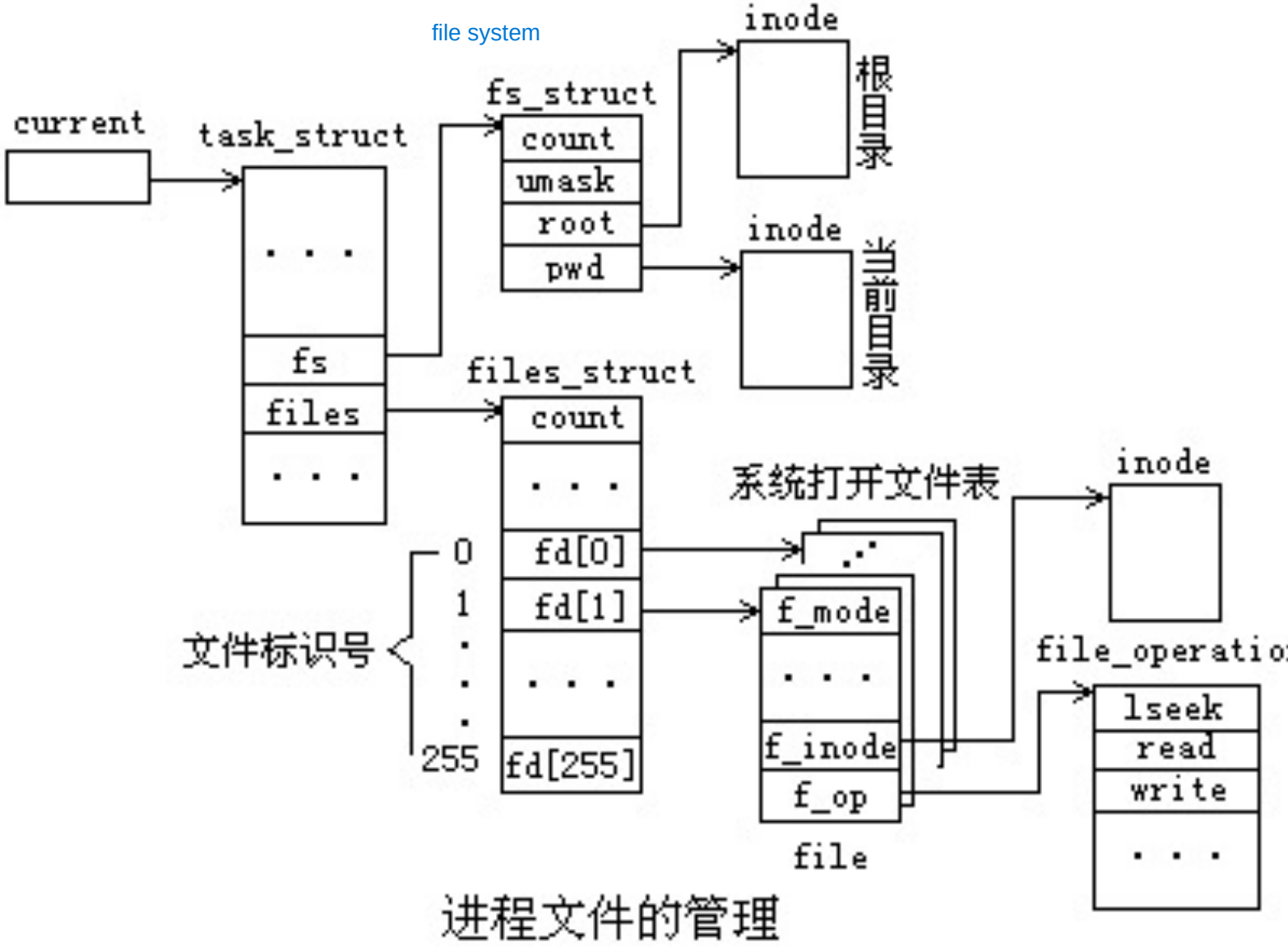
文件管理和操作

- fd[]数组的下标称为文件描述符。
- 在Linux中，进程使用文件名打开一个文件。 在此之后对文件的识别就不再使用文件名，而直接使用文件描述符。
- 在系统启动时文件描述符0、1、2由系统分配：0标准输入设备，1标准输出设备，2标准错误输出设备。
- 当一个进程通过fork()创建一个子进程后，子进程共享父进程的打开文件表，父子进程的打开文件表中下标相同的两个元素指向同一个file结构。 这时file的f_count计数值增1。
- 一个文件可以被某个进程多次打开，每次都分配一个file结构，并占用该进程打开文件表fd[]的一项，得到一个文件描述符。但它们的file结构中的f_inode都指向同一个inode。



文件管理和操作

file system



进程文件的管理

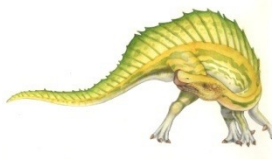
文件的open()操作

- open()系统调用内核函数是sys_open( )。

  第一个参数是打开文件的路径名。第二个参数是文件的访问标志。

sys_open( ):

- 1、Invokes getname( ) to read the file pathname from the process address space.
- 2、用get_unused_fd( )在current->files->fd所指向的文件对象指针数组中查找一个未使用的文件描述符（号），存储在局部变量fd中。
- 3、调用file_open（）函数，工作主要分成两步：
  - 第一步：调用open_namei（）函数，找到目标节点（可以是文件、目录）所对应的dentry对象，与dentry对象相对应的inode对象此时也应该在物理内存中。
  - 第二步：调用dentry_open( )函数，该函数申请一个file对象空间，然后初始化该对象，其中的一步使file对象的f_dentry指向已获得的dentry对象。



open()操作

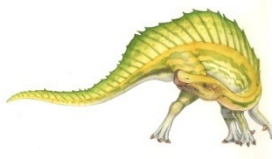
- 4、调用fd_install（）函数，将文件对象装入当前进程的打开文件表：

  current->files->fd[fd] = file;

  然后返回文件号fd。

文件目录查找

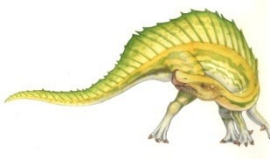
- 最重要的步骤是open_namei（）完成的。函数的执行过程如下：
  - a.确定从哪一个dentry对象出发进行路径解析。根据指定文件路径名是相对路径还是绝对路径从current中得到相应的dentry对象。
  - b.调用path_walk（）函数进行路径解析，该函数执行一个循环，每一循环都是得到一个dentry对象，该对象对应文件路径的一个子路径。



open()操作

path_walk() 由link_path_walk()函数实现，每次循环的具体过程如下：

- ① 如果子路径是“.”，表示当前目录，则应该进行下一次循环。
- ② 如果子路径是“..”，表示父目录，则做follow_dotdot动作，把dentry等对象信息更新为父目录中的数据。
- ③ 调用cache_lookup（）函数查找子路径对应的dentry是否已经在dentry cache中，如果有，则返回。
  - 若当前dentry对象已经是根目录，这时就应保持不变，直接进行下一次循环；
  - 若当前dentry对象与父目录在同一设备上，则dentry对象的d_parent成员即为所求。
  - 若当前dentry对象为所在设备的根节点，它的上一层必然是另一个设备，此时应把dentry对象改成安装点的dentry对象再从头执行② 。

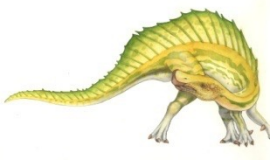


open()操作

- ④ 此时应该到磁盘上寻找，先申请dentry对象空间存放即将查找的信息，然后调用real_lookup()函数,该函数将调用父目录inode对象的i_op->lookup()方法。

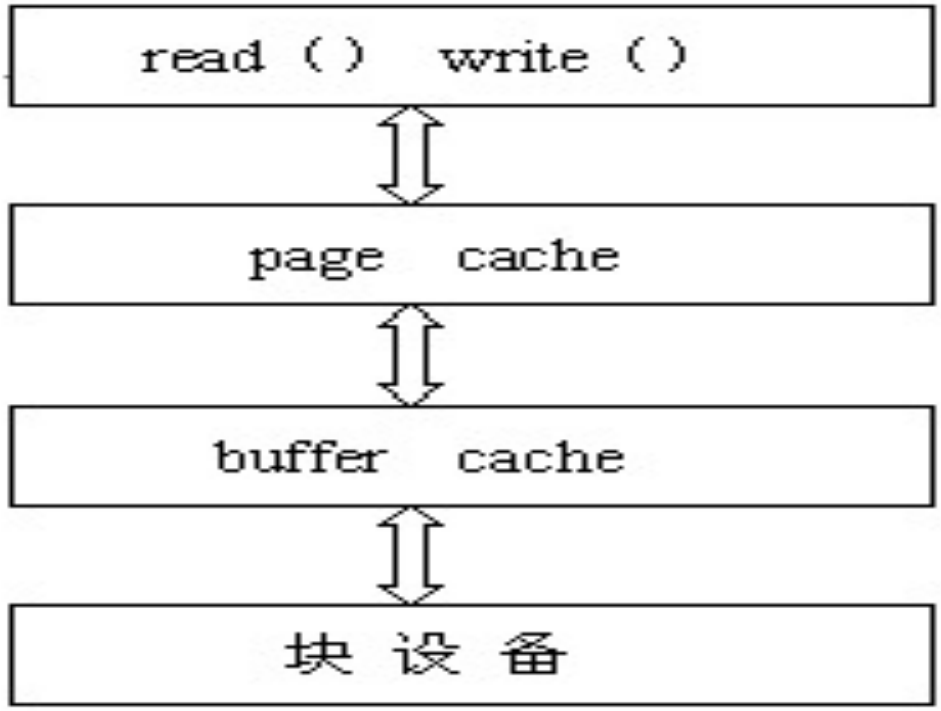
  lookup()方法是特定于文件系统的，ext2的lookup方法是ext2_lookup( )函数，它首先从磁盘读入dentry对象信息，包括dentry对象对应的inode号。然后再查找相应的inode对象是否在inode cache中，如果不在，申请一个inode对象空间再从磁盘读入信息。最后，让dentry对象的d_inode成员指向该inode对象。

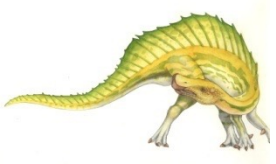
- ⑤ 已经找到子路径的dentry对象。但还有两种情况值得考虑，一是该dentry对象是一个安装点，这种情况下要推进到所安装设备的根节点。另一情况是该dentry对象是一个连接（link），这种情况下要推进到连接目标。



文件的read()操作

- 页缓冲（page cache）
  - 为了减少I/O操作的次数,Linux在读写文件时采用了页缓冲的机制。
  - 在内核中文件被看成由页面组成，对文件的读写首先要经过页缓冲。

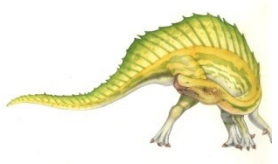




- 每个文件的所有页面由一个struct address_space 结构管理，inode结构中的i_mapping成员即起这个作用。

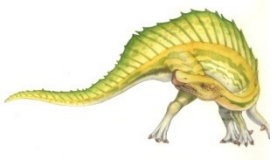
```
struct address_space {
struct list_head       clean_pages;
struct list_head       dirty_pages;
struct address_space_operations *a_ops;/*操作方法
    ......
};
struct address_space_operations {
   int (*writepage)(struct page *);
   int (*readpage)(struct file *, struct page *);
   ......
};
```

页面被链入全局hash表page_hash_table中，以加快页面的查找速度。

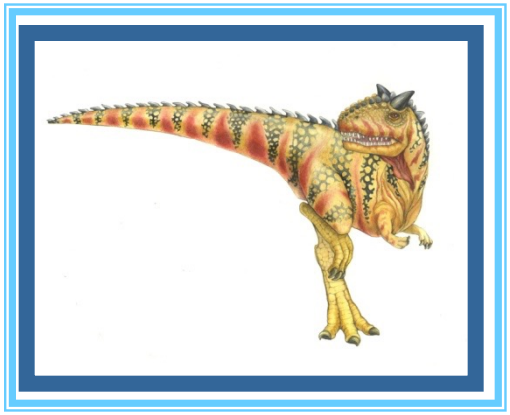


read()的实现

- read( )在内核中的对应函数为sys_read（）。
  - (1)根据文件描述符调用函数fget（），找到相应的文件对象file。
  - (2)检查file->f_mode是否允许读。
  - (3)调用file->f_op->read（）函数执行读的操作。对于大部分文件系统实际是generic_file_read( )函数。该函数根据文件位置和要读出的长度确定相应的页面，然后再检查该页面是否在pagecache中存在，如果不存在，就要调用inode节点的i_mapping->a_ops->readpage( )方法,将其从磁盘读入。不同磁盘文件系统的readpage方法不 同,ext2文件系统相应的函数为ext2_readpage。
- 为了提高性能，读操作采用了预读机制。

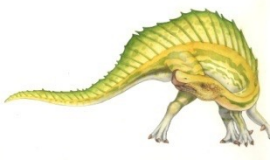


文件系统实例：ext2



ext2文件系统

- ext2的绝大多数的数据结构和系统调用与经典的UNIX一致。
- 能够管理海量存储介质。支持多达4TB的数据，即一个分区的容量最大可达4TB。
- 支持长文件名，最多可达255个字符，并且可扩展到1012个字符
- 允许通过文件属性改变内核的行为；目录下的文件继承目录的属性。
- 支持文件系统数据“即时同步”特性，即内存中的数据一旦改变，立即更新硬盘上的数据使之一致。
- 实现了“符号连接”（symbolic links）的方式，使得连接文件只需要存放inode的空间。
- 允许用户定制文件系统的数据单元（block）的大小，可以是 1024、2048 或 4096 个字节，使之适应不同环境的要求。
- 使用专用文件记录文件系统的状态和错误信息，供下一次系统启动时决定是否需要检查文件系统 。



ext2在磁盘上的物理布局

ext2分区的第一个磁盘块用于引导，其余的部分被分成组。所有的组大小相同并且顺序存放,所以由组的序号可以确定组在磁盘上的位置。

组的构成：

(1)文件系统的超级块；

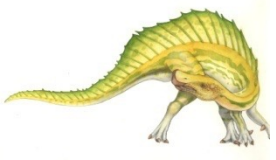
(2)所有组的描述符；

(3)数据块的位图；

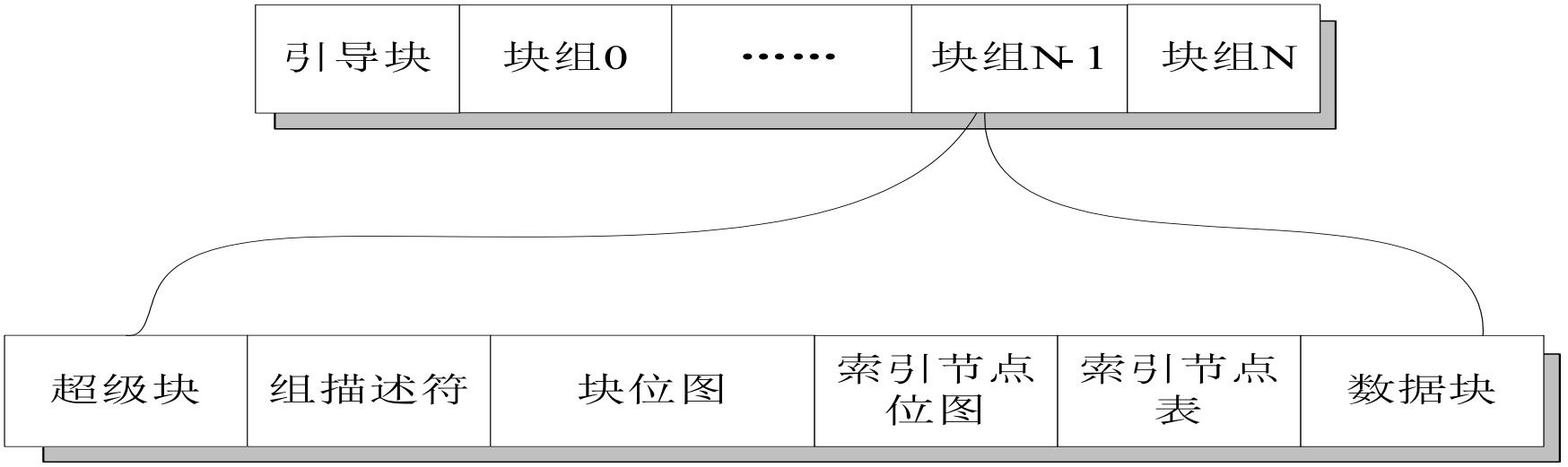
(4)inode位图；

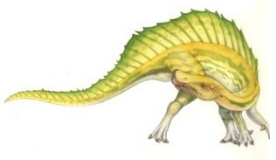
(5)inode表；

(6)数据块。



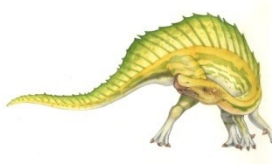
ext2体系结构





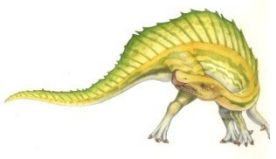
ext2体系结构

- 超级块(super block):文件系统中最重要的结构，描述整个文件系统的信息。
- 组描述符(group descriptor):记录所有块组的信息，如块组中的空闲块数、空闲节点数等。
- 数据的块位图（block bitmap）:每一个块组有一个对应的块位图。大小为一个块,位图的每一位顺序对应组中的一个块,0表示可用,1表示已用。
- inode位图（inode bitmap）：每一个块组有一个对应的inode位图。用来表示对应的inode表的空间是否已被占用。
- inode表(inode table):用来存放文件及目录的inode数据。每个文件用一个inode表示。
- include\linux\ext2.fs



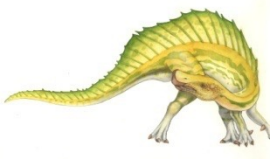
ext2超级块(super block)

- 超级块是用来描述ext2文件系统整体信息的数据结构主要描述文件系统的目录和文件的静态分布情况，以及描述文件系统的各种组成结构的尺寸、数量等。
- 超级块对于文件系统的维护是至关重要的。
- 超级块位于每个块组的最前面，每个块组中包含的超级块内容是相同的。
- 在系统运行期间，需要把超级块复制到内存的系统缓冲区内。只需把块组0的超级块读入内存，其它块组的超级块做为备份。
- 在Linux中，ext2超级块定义为ext2_super_block结构：



超级块

```
struct ext2_super_block {
 __u32  s_inodes_count; /* inode的总数 */
 __u32  s_blocks_count; /* 块的总数 */
 __u32  s_free_blocks_count;/* 空闲块的总数 */
 __u32  s_free_inodes_count;/* 空闲inode的总数 */
 __u32  s_first_data_block;        /* 第一个数据块 */
 __u32  s_log_block_size;          /* 块的大小 */
 __u32  s_blocks_per_group;        /* 每组的块数 */
 __u32  s_inodes_per_group;        /* 每组的inode数 */
 __u32  s_mtime;                   /* 安装的时间 */
 __u32  s_wtime;          /* 写的时间 */
 __u16  s_mnt_count;               /* 安装的次数 */
__u16  s_max_mnt_count;                    /* 最大可被安装的次数 */
__u16  s_magic;                   /* 文件系统标识 0x38-0x39 */
__u16  s_statemnt_count;                   /* 文件系统的状态 */
    ......
};
```



超级块

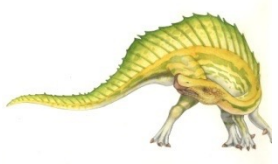
	0	1	2	3	4	5	6	7
0	inode数				块数			
8	保留块数				空闲块数			
16	空闲inode数				第一个数据块块号			
24	块长度				片长度			
32	每组块数				每组片数			
40	每组inode数				安装时间			
48	最后写入时间				安装计数		最大安装数	
56	署名		状态		出错动作		改版标志	
64	最后检测时间				最大检测间隔			
72	操作系统				版本标志			
80	UID		GID					

EXT2文件系统超级块

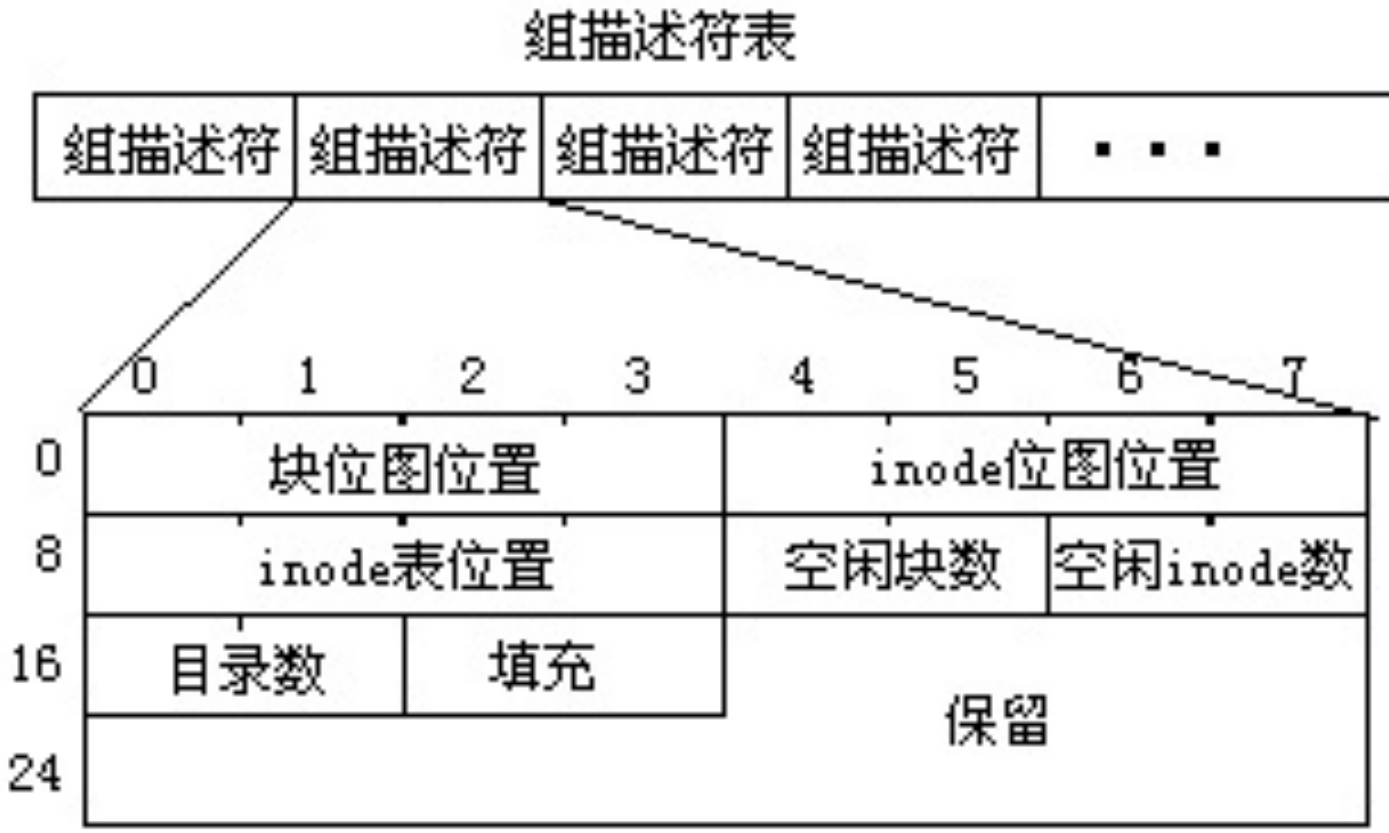
组描述符

- 每个组都有自己的描述符，用来描述一个块组的有关信息,内核用结构ext2_group_desc描述。

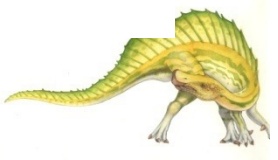
```
struct ext2_group_desc
{
  __u32 bg_block_bitmap;       /*本组数据块位图所在块号*/
  __u32 bg_inode_bitmap;       /* 本组inode位图所在块号 */
  __u32 bg_inode_table;        /* 本组inode表的起始块号 */
  __u16 bg_free_blocks_count;/* 组中空闲块的数目 */
  __u16 bg_free_inodes_count;/*组中空闲inode数目 */
  __u16 bg_used_dirs_count;/* 组中目录的数目 */
  __u16 bg_pad                  /*按字32位对齐，填充*/
  __u32 [3] bg_reserved        /*用null 填充24 个字节*/
};
```



组描述符



EXT2文件系统组描述符

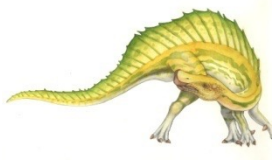


inode索引节点

- ext2文件系统inode大小为128B,inode在inode表中依次存放。
- inode表由一连串连续的块组成。 inode表第一个块的块号存放在组描述符的bg_inode_table字段中。
- 所有inode的大小相同，即128 字节。一个1024 字节的块可以包含8 个索引节点，一个4096 字节的块可以包含32 个索引节点。
- 为了计算出inode表占用了多少块，用一个组中的inode总数（存放在超级块的s_inodes_per_group字段中）除以每块中的inode数。

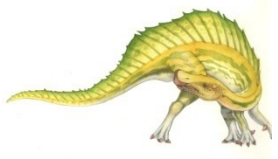
ext2_inode

```
struct ext2_inode {
        __u16  i_mode;  /* 文件类型和访问权限 */
        __u16  i_uid;  /* 拥有者的用户ID */
        __u32  i_size;/* 文件大小 */
        __u32  i_atime;       /* 最近一次访问时间 */
        __u32  i_ctime;       /* 创建时间 */
        __u32  i_mtime;       /* 最近一次修改时间 */
        __u16  i_gid;  /* 文件的组ID */
        __u32  i_blocks;/* 分配给该文件的磁盘块的数目 */
        __u32  i_block[EXT2_N_BLOCKS];/*指向磁盘块的指针
        */

        ......
};
```



inode

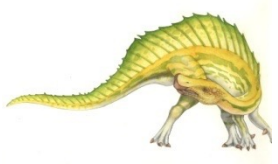
- **i_mode**：包含了文件的类型信息和文件的访问权限。文件的类型：
  - 0 未知　　4 块设备
  - 1 普通文件　　5 命名管道
  - 2 目录　　6 套接字
  - 3 字符设备　　7 符号链接
- **i_block[EXT2_N_BLOCKS]**：一般用于放置文件的数据所在的磁盘块编号,EXT2_N_BLOCKS的默认值为15。
  - 不同类型文件的i_block的含义不同：
    - 设备文件：用ext2_inode就足以包含所有信息,不需要另外的数据块。
    - 目录：数据块包含了所有属于这个目录的文件信息。数据块的数据项是个ext2_dir_entry_2结构,用来描述属于这个目录的文件。 目录的各项在数据块中依次放置。



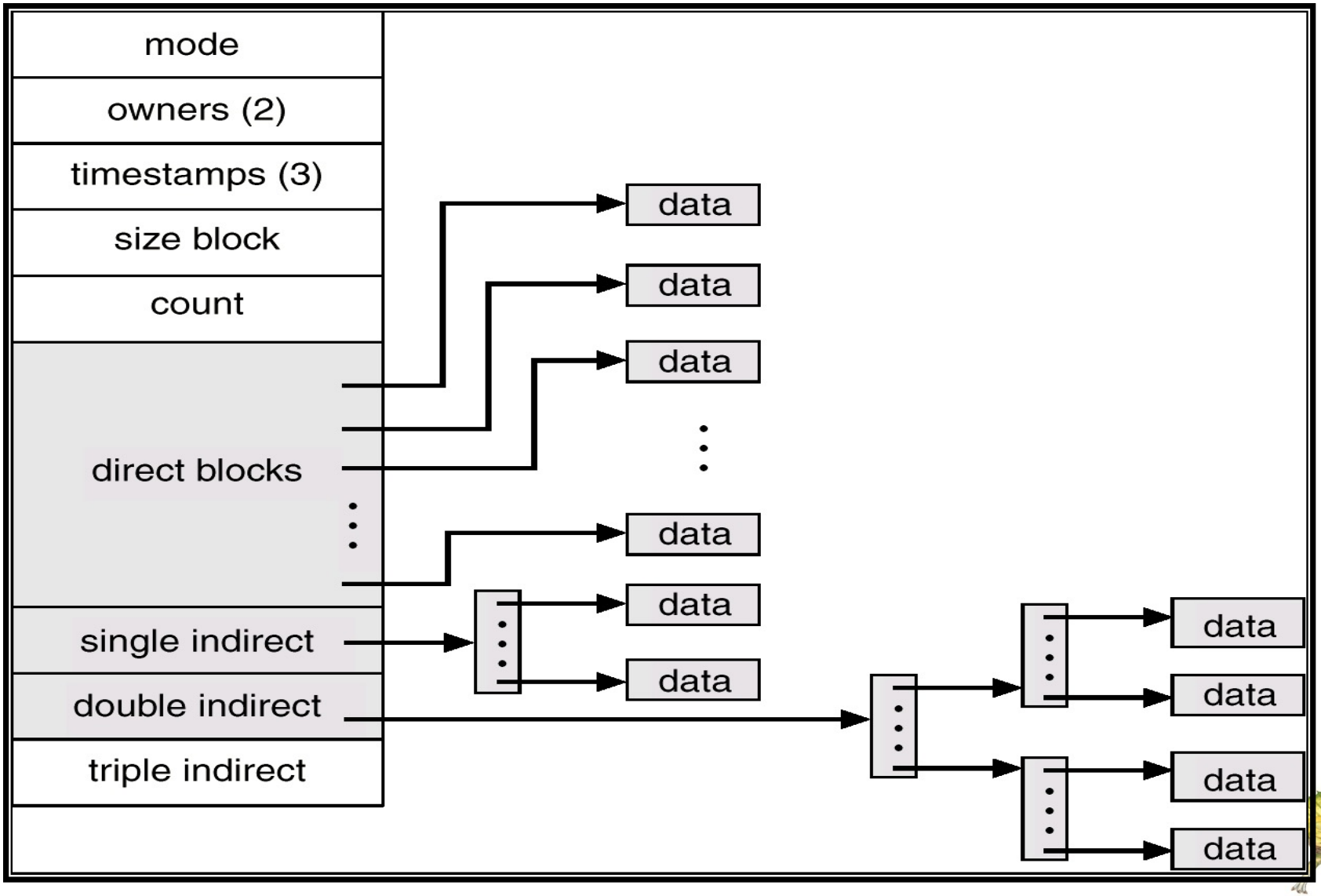
inode

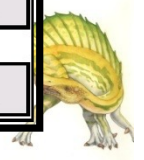
```
struct ext2_dir_entry_2 {
  __u32 inode;  /* inode号 */
  __u16 rec_len;        /* 本项所占的长度 */
  __u8  name_len;       /* 文件名长度 */
  __u8  file_type; /* 文件类型 */
  char  name[EXT2_NAME_LEN];       /* 文件名 */
};
```

- 普通文件：指向数据块，数据在磁盘上并不一定连续,需要保存各个磁盘块号。i_block[]的前12项可看成一级指针,直接存放文件数据所在的磁盘块号。第13项是个二级指针（一级索引），第14,15项分别是三级指针和四级指针（二级、三级索引）。

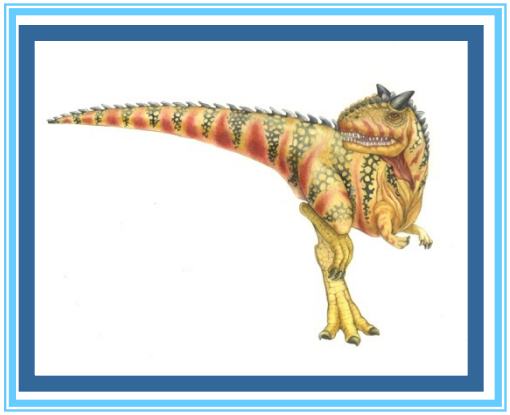


- 这种方法保证了大量的小文件访问效率高，同时又支持大文件。





实验：添加一个文件系统



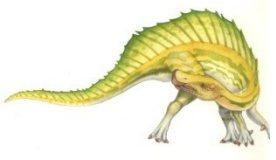
实验中用到的命令

- dd：用指定大小的块拷贝一个文件，并在拷贝的同时进行指定的转换
- 命令语法：dd [选项]
- 常用参数：
  - if = 输入文件（或设备名称）
  - of = 输输出文件（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： dd if=/dev/zero of=myfs bs=1M count=1

◆/dev/zero：零设备“0”
◆/dev/loop：loopback device（回环设备、或虚拟设备）指是用文件来模拟块设备

实验：添加一个文件系统

- myext2文件系统的物理格式定义与ext2基本一致，除了myext2的magic number是0x6666，而ext2的magic number是0xEF53
- myext2是ext2的定制版本，它只支持原来ext2文件系统的部分操作，以及修改了部分操作

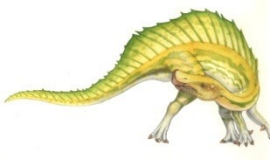


1. 添加一个文件系统

- 把ext2部分的源代码和结构克隆为myext2

  # cd ~/linux-3.11 /* 内核源代码目录，假设内核源代码解压在主目录的Linux-3.11只目录中*/

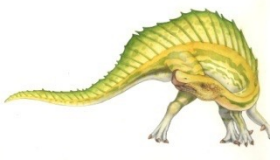
- fs/ext2/balloc.c
- fs/ext2/bitmap.c
- fs/ext2/dir.c
- fs/ext2/file.c
- fs/ext2/fsync.c
- fs/ext2/ialloc.c
- fs/ext2/inode.c
- fs/ext2/ioctl.c
- fs/ext2/namei.c
- fs/ext2/super.c
- fs/ext2/symlink.c
- include/linux/ext2_fs.h
- include/linux/ext2_fs_i.h
- include/linux/ext2_fs_sb.h



添加一个文件系统

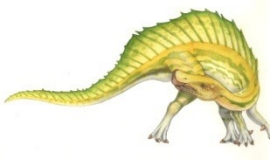
修改：

```
#!/bin/sh
SCRIPT=substitute.sh
for f in *;
do
    if [ $f = $SCRIPT ]; then
        echo "skip $f"
        continue
    fi
    echo -n "substitute ext2 to myext2 in $f..."
    cat $f | sed 's/ext2/myext2/g' > ${f}_tmp
    mv ${f}_tmp $f
    echo "done"
    echo -n "substitute EXT2 to MYEXT2 in $f..."
    cat $f | sed 's/EXT2/MYEXT2/g' > ${f}_tmp
    mv ${f}_tmp $f
    echo "done"
done
```



添加一个文件系统

- linux/include/linux/fs.h
- ……
- struct ext2_inode_info ext2_i;
- struct xs_ext2_inode_info myext2_i;
- ……
- struct ext2_sb_info ext2_sb;
- struct xs_ext2_sb_info myext2_sb;
- ……

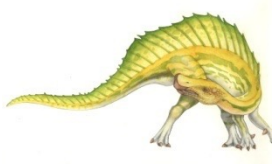


添加一个文件系统

- 添加：
  - 在include/uapi/asm-generic/bitops.h文件中添加：
    - kernel 3.11 :

```
#include <asm-generic/bitops/myext2-atomic.h>
```

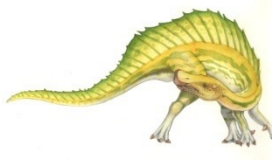
  - 在include/linux/magic.h 文件中添加”#define MYEXT2_SUPER_MAGIC 0xEF53”



添加一个文件系统

修改配置文件：

- 在fs/Kconfig文件中增加 source “fs/myext2/Kconfig”，并且对“ext2”相关项的地方添加“myext2”项。
- 在 fs/Makefile 文件中添加“obj-$(CONFIG_MYEXT2_FS) += myext2/”
- 为了在make mencuconfig中看得更加清楚，修改fs/myext2/Kconfig文件中“Ext2”替换为“MYExt2”, My Second extended fs support => My MYSecond …



添加一个文件系统

- 编译新内核
- 重新启动
- 测试一下
  - dd if=/dev/zero of=myfs bs=1M count=1
  - mkfs.ext2 myfs
  - cat /proc/filesystems | grep ext

    ext2

    ext3

    myext2

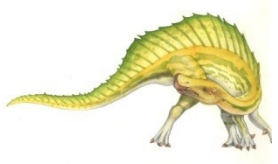
    ext4
  - mount –t myext2 –o loop ./myfs /mnt/myext2
  - mount

    …..

    /dev/loop0 on /mnt type myext2 (rw)
  - umount /mnt
  - mount –t ext2 –o loop ./myfs /mnt/myext2
  - mount

    …..

    /dev/loop0 on /mnt type ext2 (rw)



2. 修改myext2的magic number

- 修改include/linux/magic.h文件：

  #define MYEXT2_SUPER_MAGIC 0xEF53

  改为 #define MYEXT2_SUPER_MAGIC 0x6666

- 编译新内核
- 重新启动
- 测试一下
  - dd if=/dev/zero of=myfs bs=1M count=1
  - mkfs.ext2 myfs
  - ./changeMN myfs
  - mount –t myext2 –o loop ./fs.new /mnt
  - mount

    /dev/loop0 on /mnt myext2 (rw)
  - umount /mnt
  - mount –t ext2 –o loop ./fs.new /mnt

    mount: wrong fs type, bad option, bad superblock on /dev/loop0, …

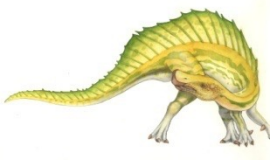
3. 修改文件系统操作

这一步可以与第2步一起修改，一次编译

- 裁减myext2的mknod操作
- 修改完毕，然后重新编译内核。以新生成的内核重新启动计算机，我们在shell下执行如下测试程序：
  - mount –t myext2 –o loop ./fs.new /mnt
  - cd /mnt
  - mknod myfifo p

    haha, mknod is not supported by myext2! You've been cheated!

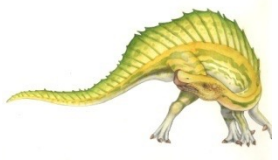
    mknod: `myfifo': Operation not permitted



4. 添加文件系统创建工具

- 创建文件系统的程序：
  - 编写mkfs.myext2

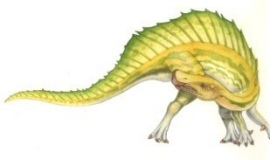
```
#!/bin/bash

/sbin/losetup -d /dev/loop0
/sbin/losetup /dev/loop0 $1
/sbin/mkfs.ext2 /dev/loop0
dd if=/dev/loop0 of=./tmpfs bs=1k count=2
./changeMN ./tmpfs
dd if=./fs.new of=/dev/loop0 bs=1k count=2
/sbin/losetup -d /dev/loop0
rm -f ./tmpfs
```



mkfs.myext2

- 测试一下
  - dd if=/dev/zero of=myfs bs=1M count=1
  - mkfs.myext2 myfs
  - mount –t myext2 –o loop ./fs.new /mnt/myext2
  - mount

    /dev/loop0 on /mnt myext2 (rw)



作业

- 根据教材，完成“添加一个文件系统”实验
- 分析open系统调用代码，P. 608-611， P. 592-600