

*Linux* 源码分析 系列之

# 文件系统

唐欢

htang@ncic.ac.cn

智能中心

2004.4

# 概要

---

- 静

- UNIX系统V文件系统
- Ext2文件系统

- 动

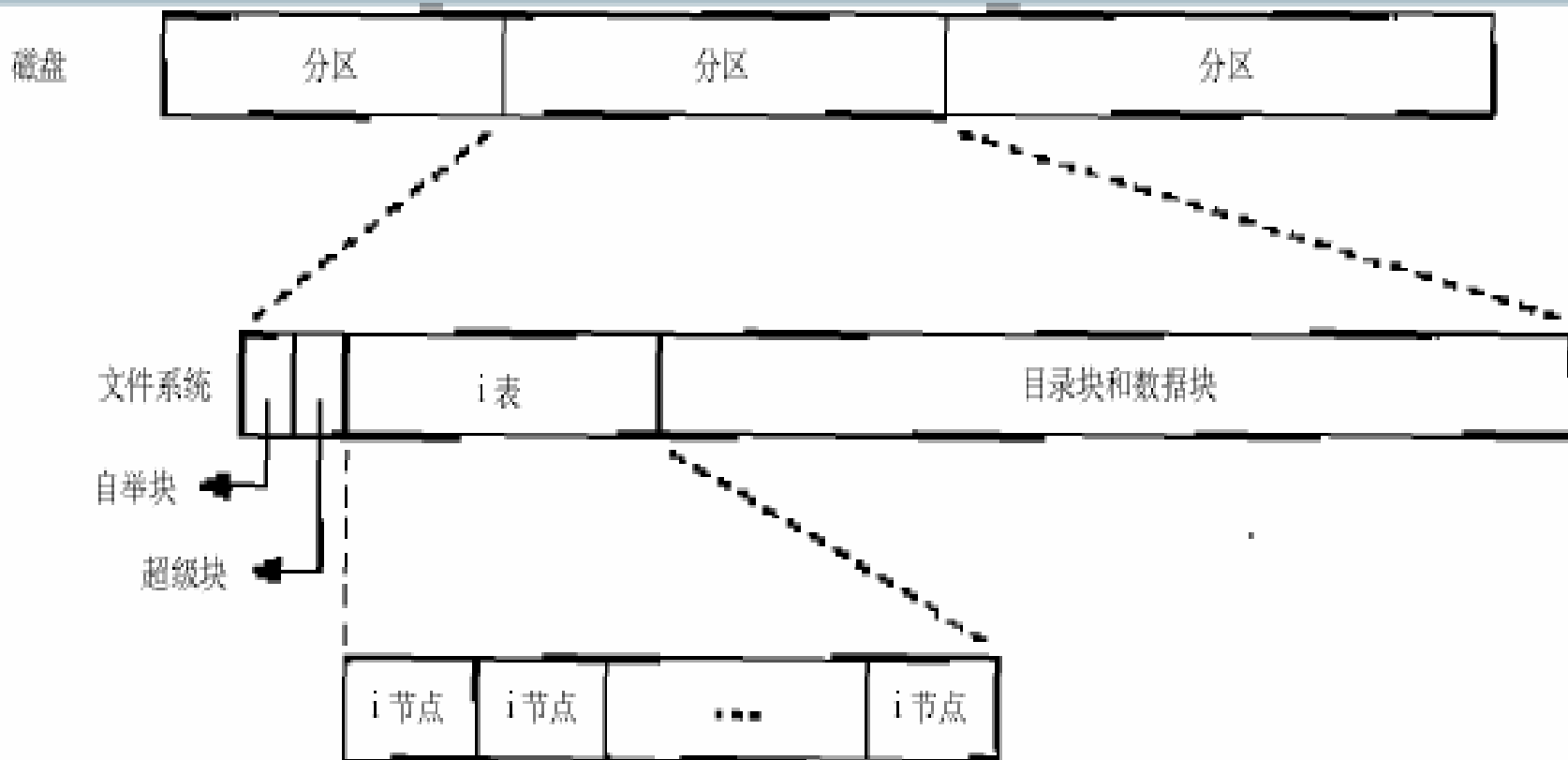
- VFS文件系统

- 静↔动

- 文件系统操作—mount root filesystem

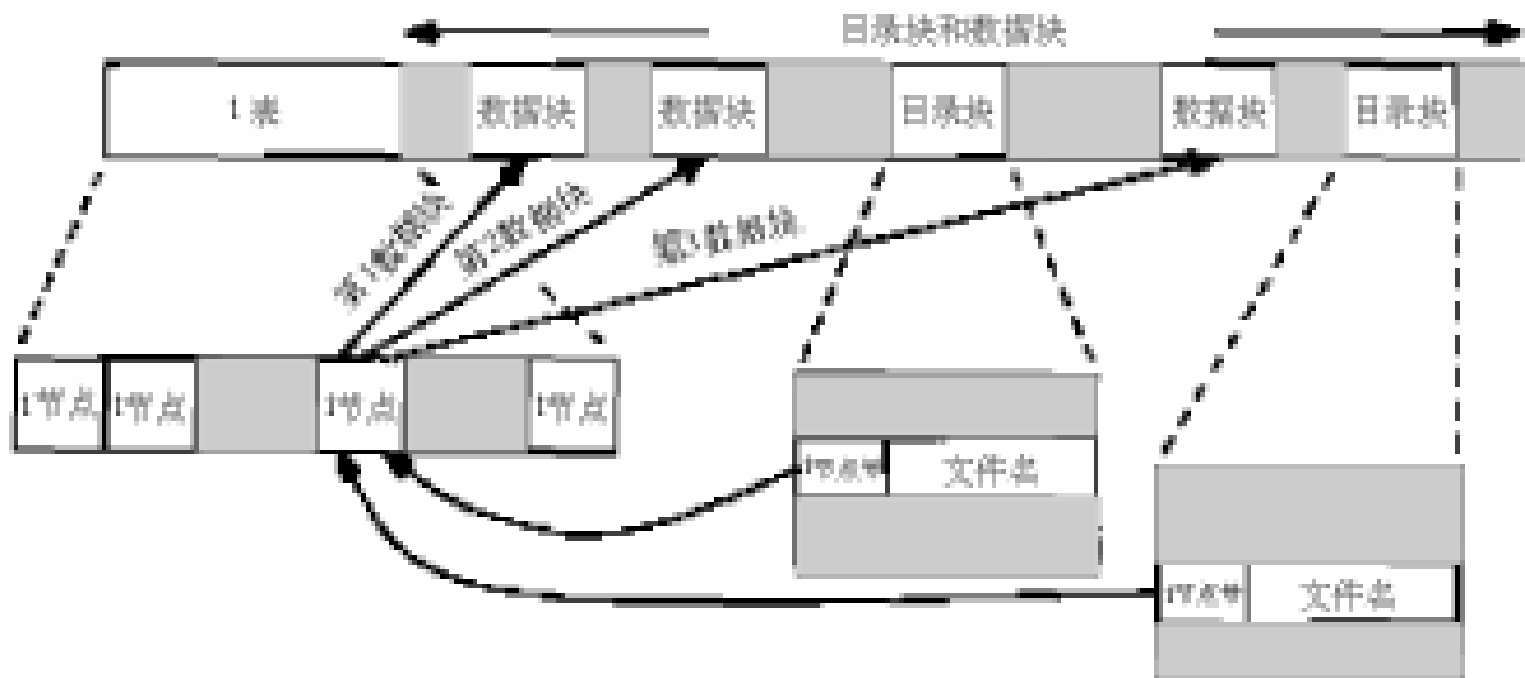
# UNIX系统V文件系统

# UNIX系统V文件系统



# UNIX系统V文件系统

## 文件的信息都放在哪里？



- **讨论点**：文件名与inode的分离。为什么这样设计？还可以怎么样设计？

# UNIX系统V文件系统

## 从目录项到inode

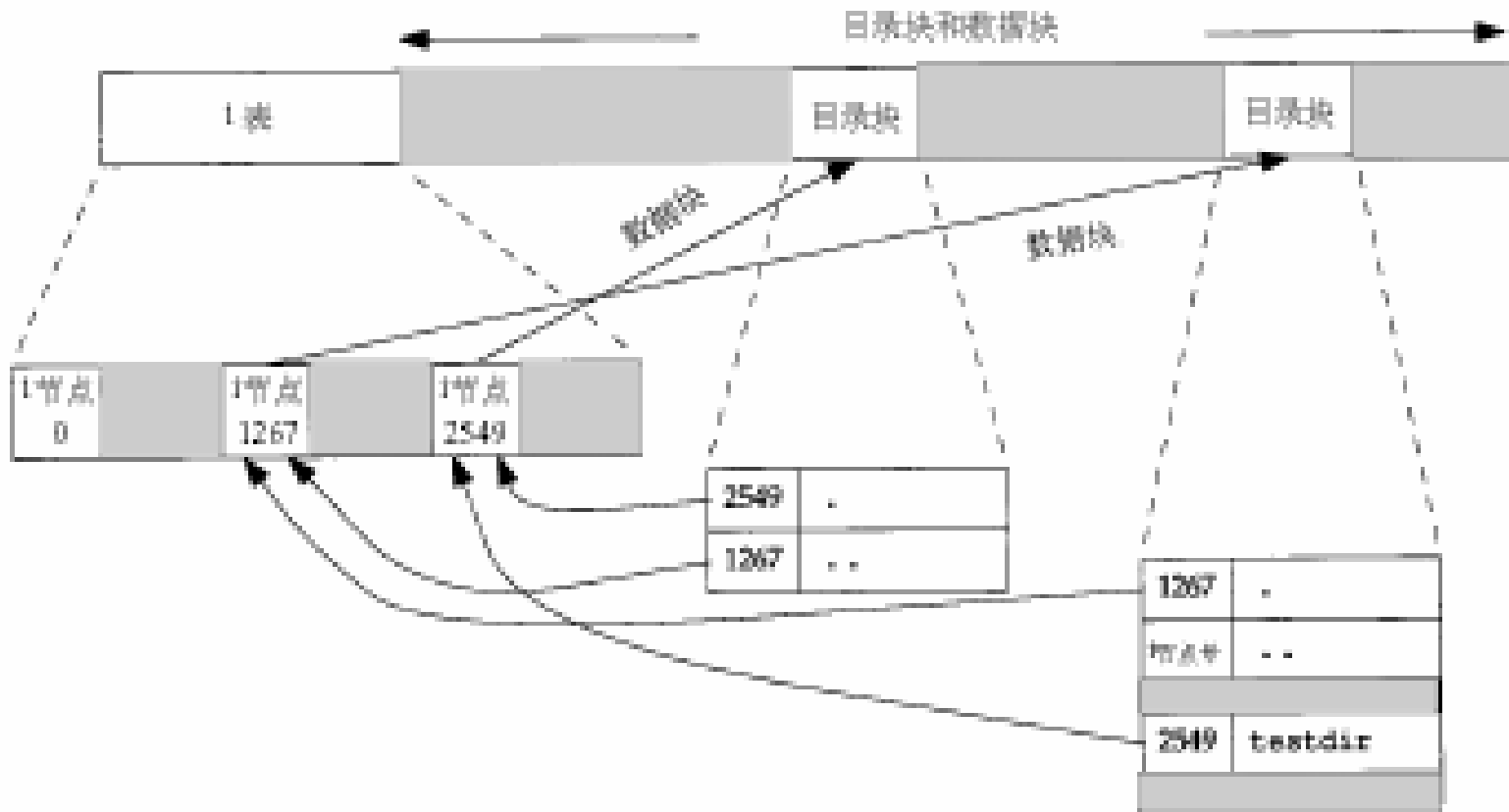
---

- 目录项中的inode编号指向“同一文件系统”中的inode，因此企图将目录项中inode指向其他文件系统的操作都是不允许的。例如：
- ln命令将建立一个新的目录项，指向一个现存的文件。当 /public 为NFS时：

```
dcos3:~ # ln -d /public/ /tmp/pub  
ln: creating hard link `/tmp/pub' to `/public':  
Invalid cross-device link
```

- **讨论点**：此举受到什么限制，能否突破？突破又有何意义？

## 目录怎样认识自己和爸爸



---

# Ext2文件系统



# Ext2文件系统

以block(块)为基本单位，对磁盘进行划分。block的大小在创建文件系统时可以指定。

n blocks。

保存了本组所有的inode。应inode描述文件与目录，一个文件对应一个inode，目录是一种特殊的文件。

inode 不仅记录了文件内容在外存空间的位置，而且定义了文件的存取权限、修改时间、文件类型。

## Block Group

将连续的block分组。为什么？

尽量将属于一个文件的block存放于一个组内，保证了访问局部性。

Super  
Block

D

1 block

n block

1 block

1 block

11 blocks

11 blocks

# Ext2文件系统

## 多少个block group ?

---

- 两个决定因素：分区大小 & block大小
- group数量 = 分区大小 / group大小
  - group大小 = block大小 X block数量
  - block数量 = block bitmap 位数(bit)
  - 且block bitmap 只占用一个block ,则：
  - block bitmap 位数 = block大小 X 8
- group大小 = block大小<sup>2</sup> X 8

# Ext2文件系统

## Super Block (/include/linux/Ext2\_fs.h)

---

- 记录super block在磁盘上的结构
- struct ext2\_super\_block {
  - 每一个block有多大？
  - 我有多少Inode？多少block？他们中多少是free的？
  - 多少个block被保留？
  - 每个block group中，有多少个block？多少个inode？
  - 第一个Data block在哪里？
  - 我何时mount？mount了几次？最多可以mount几次？
  - 上一次write操作在何时？上一次check操作在何时？
  - .....
  - }

# Ext2文件系统

## Group Descriptor (/include/linux/Ext2\_fs.h)

---

- struct ext2\_group\_desc
- {
- \_\_u32 bg\_block\_bitmap; /\* Blocks bitmap block \*/
- \_\_u32 bg\_inode\_bitmap; /\* Inodes bitmap block \*/
- \_\_u32 bg\_inode\_table; /\* Inodes table block \*/
- \_\_u16 bg\_free\_blocks\_count; /\* Free blocks count \*/
- \_\_u16 bg\_free\_inodes\_count; /\* Free inodes count \*/
- \_\_u16 bg\_used\_dirs\_count; /\* Directories count \*/
- \_\_u16 bg\_pad;
- \_\_u32 bg\_reserved[3];
- }

# Ext2文件系统

## Inode —基本“构件”

- struct ext2\_inode {
- \_\_u16 i\_mode; ←
- \_\_u32 i\_size;
- \_\_u32 i\_atime; ←
- \_\_u32 i\_ctime; ←
- \_\_u32 i\_mtime; ←
- \_\_u32 i\_dtime;
- \_\_u16 i\_links\_count;
- \_\_u32 i\_blocks; ↘
- \_\_u32 i\_block[EXT2\_N\_BLOCKS];
- \_\_u32 i\_file\_acl; ←
- \_\_u32 i\_dir\_acl; ←
- }

i\_mode

文件模式 表示文件类型

atime ctime mtime

atime-- 文件数据的最后访问时间

mtime-- 文件数据的最后修改时间

i\_dtime-- 文件数据的最后修改时间

i\_block

指向inode所描述的Data  
block的指针。其中：  
EXT2\_N\_BLOCKS = 15

acl

Access Control List

/\* Directory ACL \*/

# Ext2文件系统

## i\_mode 表示的多种文件类型

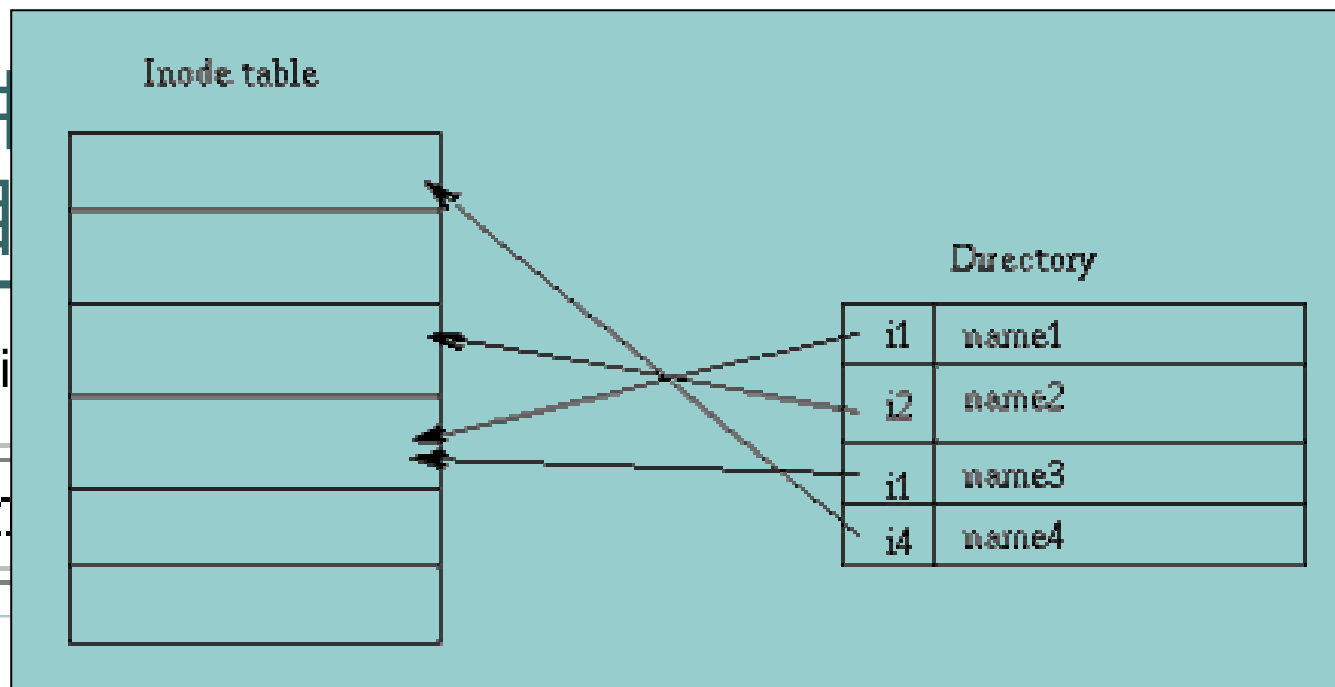
File_type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

# Ext2文件 EXT2 目

- (/include/li

inode r

Ex.



	inode			rec_len		name							
0		21		12	1 2	-	\0	\0	\0				
12		22		12	2 2	.	.	\0	\0				
24		53		16	5 2	h	o	m	e	1	\0	\0	\0
40		67		28	3 2	u	s	r	\0				
52		0		16	7 1	o	l	d	f	i	l	e	\0
68		34		12	4 2	s	b	i	n				

```
__u32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
```

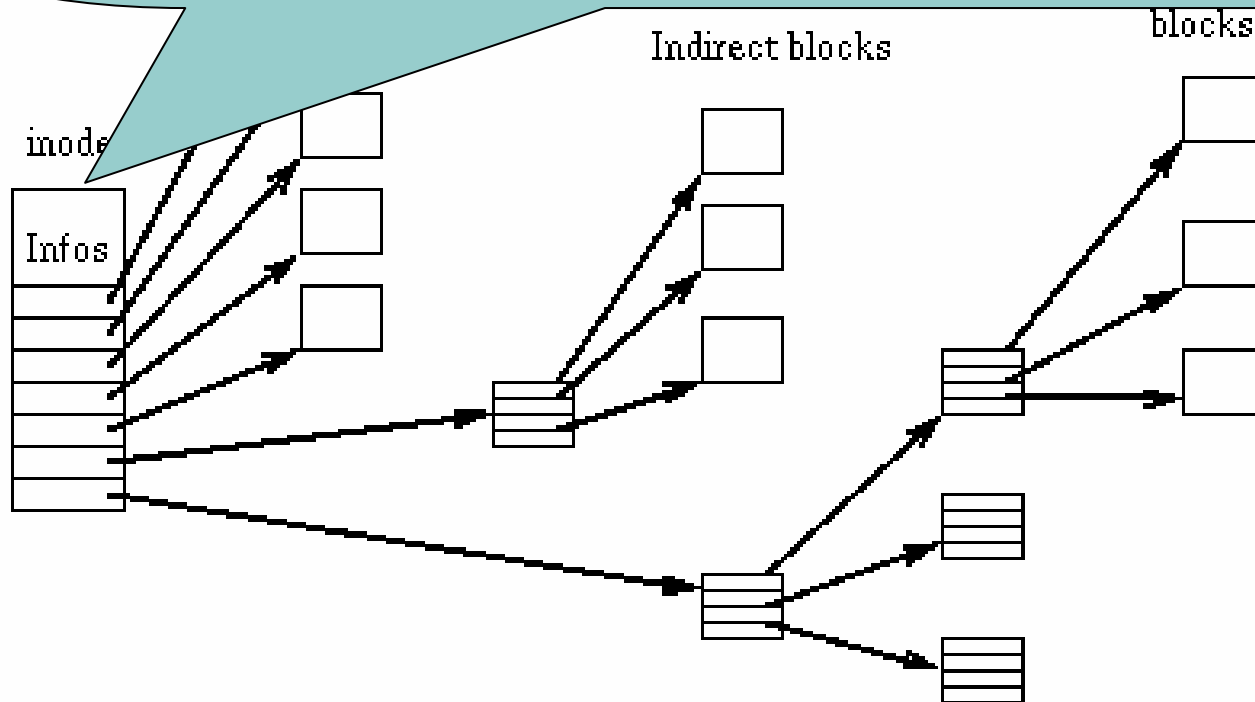
```
#define EXT2_NDIR_BLOCKS 12
```

```
#define EXT2_IND_BLOCK EXT2_NDIR_BLOCKS //12
```

```
#define EXT2_DIND_BLOCK (EXT2_IND_BLOCK + 1) //13
```

```
#define EXT2_TIND_BLOCK (EXT2_DIND_BLOCK + 1) //14
```

```
#define EXT2_N_BLOCKS (EXT2_TIND_BLOCK + 1) //15
```





# Ext2文件系统

## 符号连接 Symbolic link

---

- 符号连接记录了目标的路径。如果这一“记录”的长度小于 60 字符, 将被写入 i\_block 数组中。否则, 当其长度大于60时, 将占用一个block来存储。
- 60 从哪里来?

`__u32 i_block[EXT2_N_BLOCKS];`

4 Byte

15

i\_block占用空间 = 15 X 4 = 60 Byte

# Ext2文件系统

## 谁不需要block ?

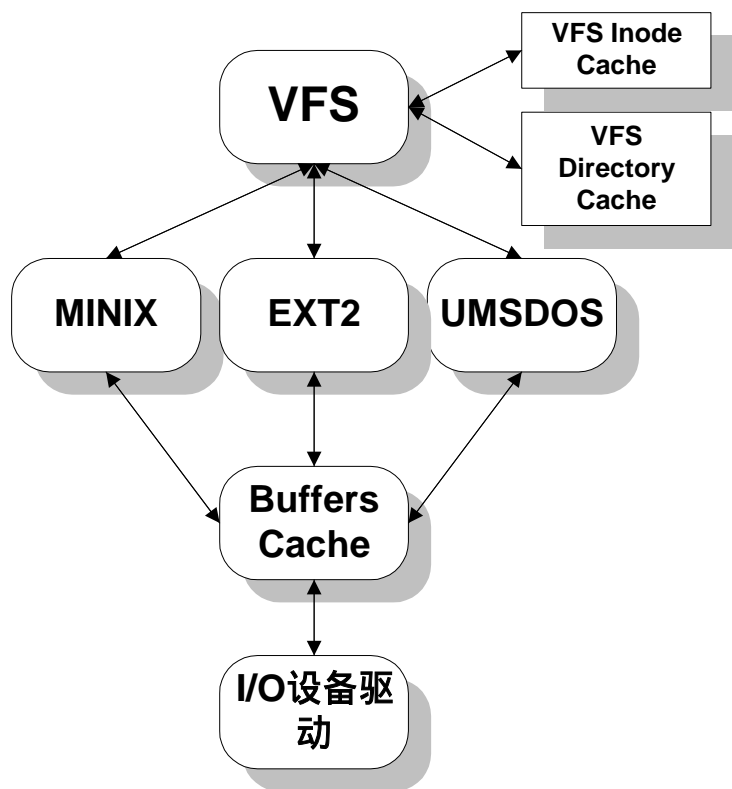
---

- 空文件
- 符号连接(当目标路径长度  $< 60$  )
- 设备文件(块设备、字符设备)
- Socket
- Named pipe
- ?

---

# VFS文件系统

# VFS文件系统



- VFS是物理文件系统与服务之间的一个**接口层**。
- VFS对Linux的每个文件系统的所有细节进行**抽象**。
- VFS只存在于**内存中**，不存在于任何外存空间。
- VFS在系统启动时**建立**，在系统关闭时**消亡**。

# VFS文件系统

MM → VFS

---

物理内存可动态地映射成进程的虚拟内存

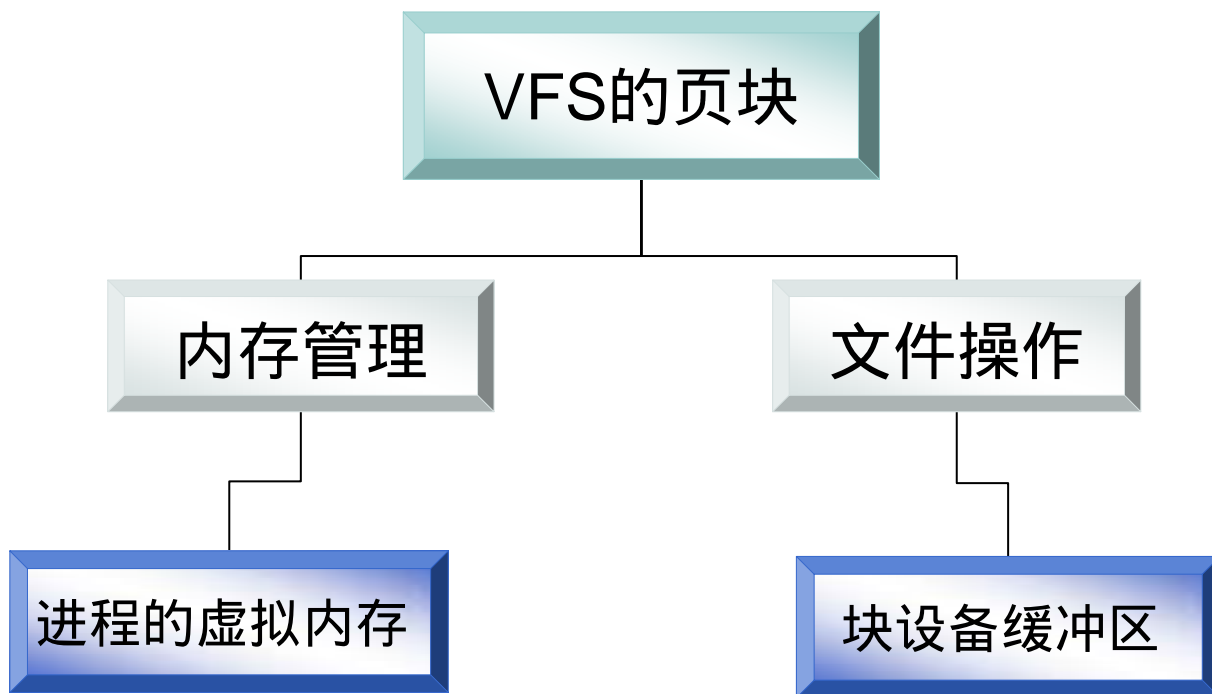
外部文件系统可动态地映射成VFS

外部文件系统中的文件可动态地映射成VFS中的文件。

说得更精确些，就是外部文件系统中某一文件的某一物理块可以动态映射成VFS中该文件的页块。

# VFS文件系统 功用

---



# VFS文件系统

## 为文件操作提供的函数

---

- 管理缓冲区 (buffer.c)
- 响应系统调用fcntl() 和ioctl() (fcntl.c & ioctl.c)
- 将管道和文件输入/输出映射到索引节点和缓冲区(fifo.c, pipe.c)
- 锁定和不锁定文件和记录(locks.c)
- 映射名字到索引节点(namei.c, open.c)
- 实现select( )函数(select.c)
- 提供各种信息(stat.c)
- 挂接和卸载文件系统(super.c)
- 调用可执行代码和转存核心(exec.c)
- 装入各种二进制格式(bin\_fmt\*.c)

# VFS 文件系统

## 主要三个进程打开

### inode object

保存某一文件的基本信息。  
For disk-based filesystems, this object usually corresponds to a file control block stored on disk.

### object

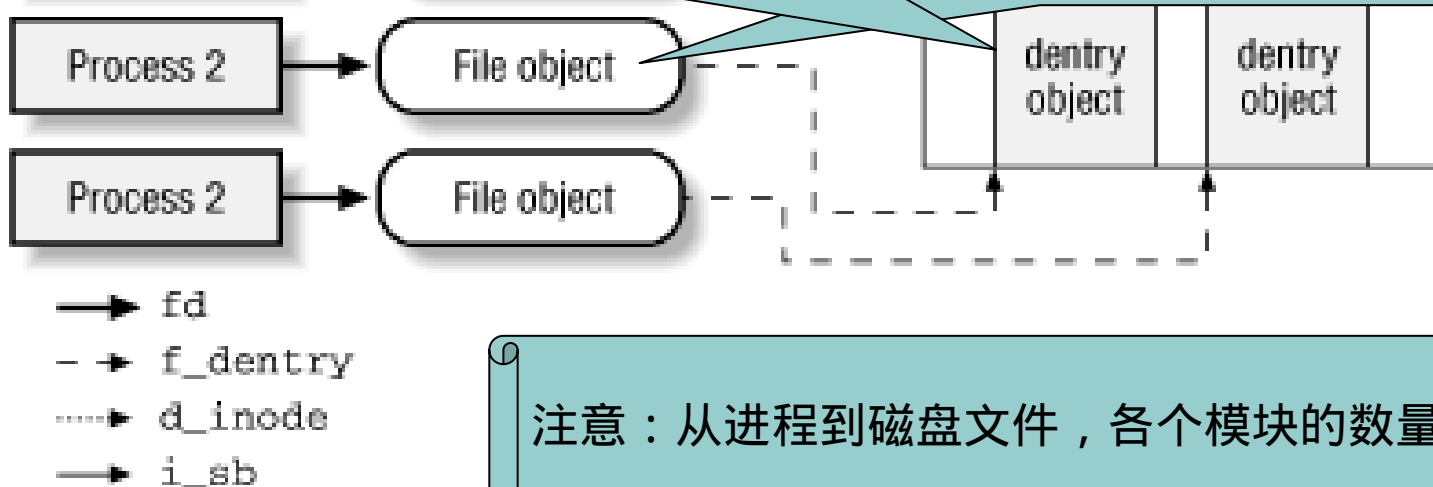
保存文件相关的信息。  
For disk-based filesystems, this object usually corresponds to a file control block stored on disk.

### dentry object

保存目录项和其相应文件间链接的信息。  
不同的文件系统以不同方式存储这种信息。  
ext2: 目录是特殊的文件  
FAT (File Allocation Table) : 目录不是文件, 使用FAT记录文件在目录树中的位置。

### object

保存文件的交互信息。这  
是在访问文件时存储与内



注意：从进程到磁盘文件，各个模块的数量



# 编外—源码中面向对象的痕迹

- VFS的OO特性
  - VFS是对所有具体文件系统的抽象。包括系统的结构与相关的操作。
- VFS的OO实现

Object XXX
数据成员
成员函数

VFS `Object`
struct XXX{ struct XXX_operations * X_op; }
struct XXX_operations{ (*function_A) (*function_B) }

# VFS文件系统

## super block Object (/include/linux/fs.h)

### super\_block Object

```
struct super_block {  
    struct super_operations *s_op;  
    ....  
}  
  
struct super_operations {  
    void (*read_inode) (struct inode *);  
    void (*write_inode) (struct inode *, int);  
    void (*delete_inode) (struct inode *);  
    ....  
}
```

为保证文件系统性能，设备的super block必须驻留内存，整个VFS的super block实现这样的内存空间。

super block记录文件系统的具有共性的信息(如:文件锁、修改标志)，还连接着具体文件系统的特有信息。同时，拥有着作用于这些信息成员之上的操作。

# VFS文件系统

## super block 数据成员 (/include/linux/fs.h)

### super\_block Object

```
struct super_block {
```

```
    struct list_head s_list;
```

```
    struct file_system_type *s_type;
```

```
    unsigned char s_dirt;
```

```
    struct super_operations *s_op;
```

```
    struct dentry *s_root;
```

```
    union{....} u;
```

```
    ....
```

```
}
```

super\_operations

super block部分数据成员：

连接多个super block的双向链表

文件系统类型

已修改(脏)的标志

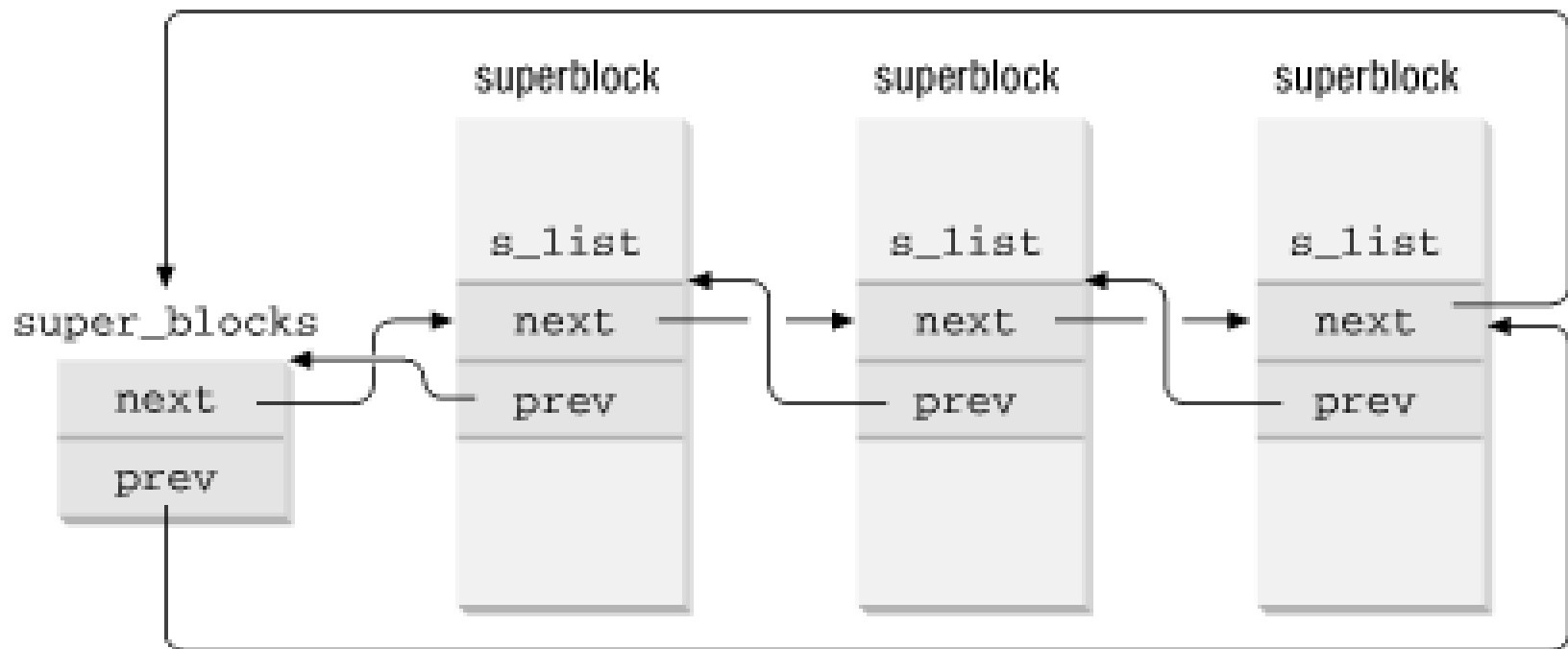
super block 操作集

挂载的dentry object 对象

特定文件系统的super block信息。

# VFS文件系统

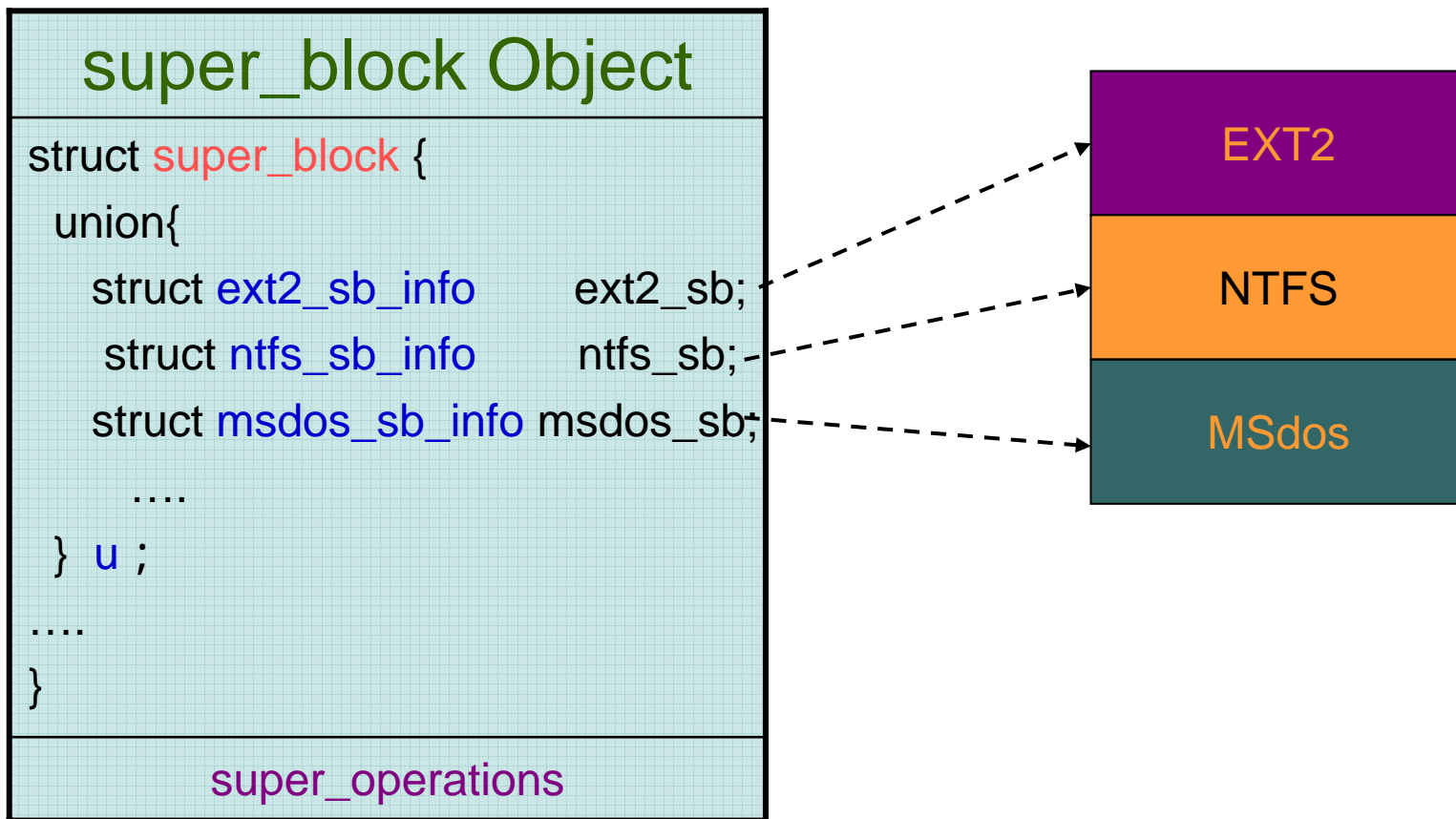
super block 数据成员 `super_block.s_list`



**讨论点：**还有很多的这样的链表，连在一起组成一张网，有谁清楚知道网的结构？有什么共性？

# VFS文件系统

super block 数据成员 super\_block.u



# VFS文件系统

## super block 函数成员 (/include/linux/fs.h)

### super\_block Object

super\_block

```
struct super_operations {
```

```
    read_inode (inode)
```

```
    dirty_inode (inode)
```

```
    write_inode (inode, flag)
```

```
    put_inode(inode)
```

```
    delete_inode(inode)
```

```
    clear_inode(inode)
```

```
    ....
```

```
}
```

- superblock函数成员功能：
- 从磁盘读取信息填写inode(内存)
- 当dirty标志被置位时执行(EXT3)
- 更新inode内容，flag决定是否同步磁盘
- 释放与inode关联，不释放内存
- 删除inode(磁盘操作)
- 同put\_inode,同时释放内存

# VFS文件系统

## inode Object (/include/linux/fs.h)

### inode Object

```
struct inode {  
    struct inode_operations *i_op;  
    ....  
}
```

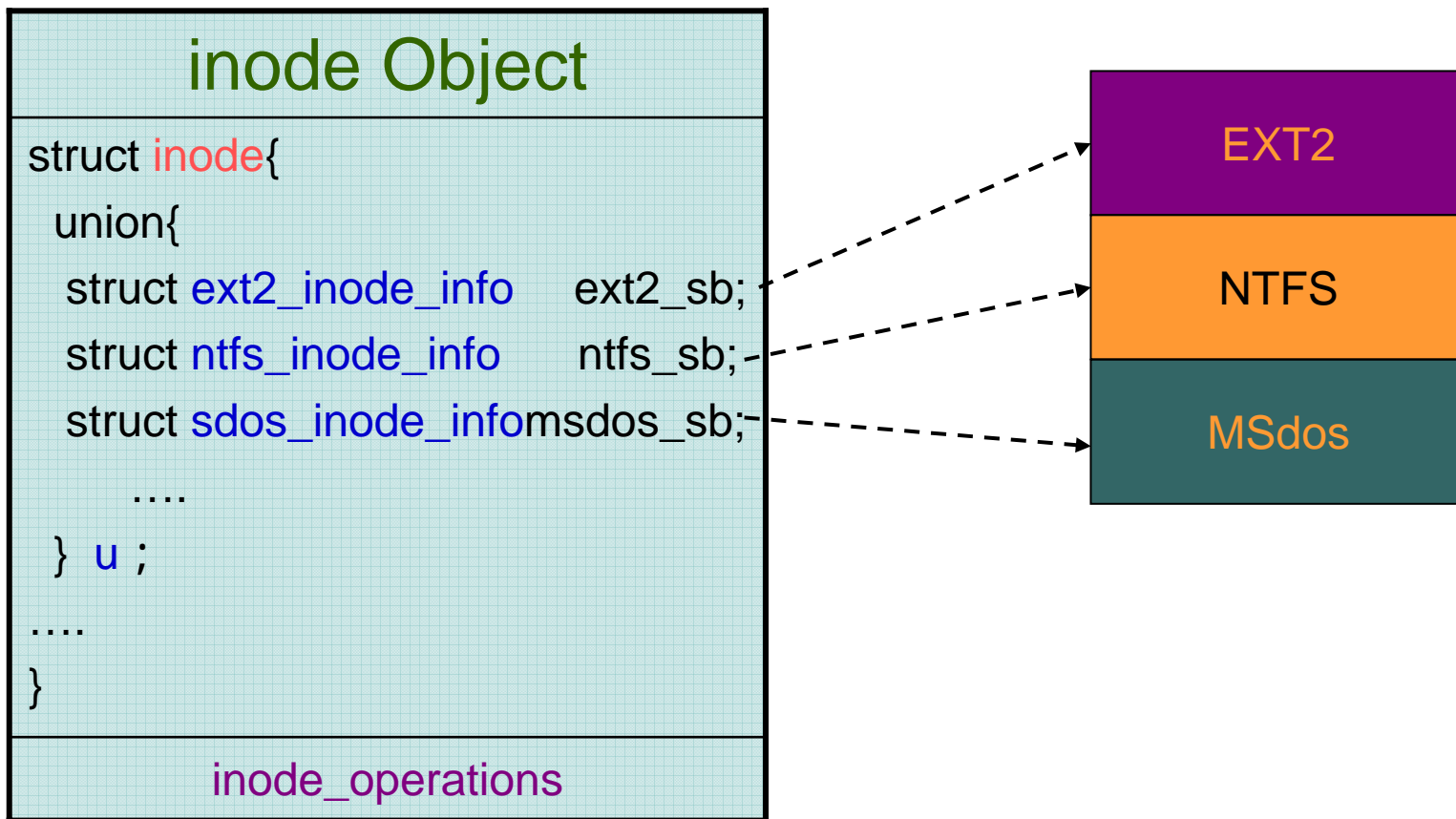
```
struct inode_operations {  
    int (*create)  
    int (*link)  
    int (*mkdir)  
    ....  
}
```

文件系统对文件和目录的控制全都由 inode 实现。inode 是 Linux 管理文件系统的最基本的单位，也是任何文件系统连接任何子目录、任何文件的桥梁。

VFS 的 inode 对数据与操作的实现与 super block 相同。

# VFS文件系统

## inode 数据成员 inode.u





# VFS文件系统

## inode 函数成员 (/include/linux/fs.h)

---

### inode Object

inode\_block

```
struct inode_operations {  
    create(dir, dentry, mode)  
    lookup(dir, dentry)  
    link(old_dentry, dir, new_dentry)  
    unlink(dir, dentry)  
    symlink(dir, dentry, symname)  
    mkdir(dir, dentry, mode)  
    rename(old_dir, old_dentry,  
          new_dir, new_dentry)  
    ....  
}
```

# VFS文件系统

## file Object (/include/linux/fs.h)

### file Object

```
struct file {  
    struct file_operations    *f_op;  
    ....  
}
```

```
struct file_operations {  
    loff_t (*llseek)  
    ssize_t (*read)  
    ssize_t (*write)  
    ....  
}
```

- file Object 描述了进程与其打开文件间的交互过程。
- file Object在进程打开一个文件时创生。

# VFS文件系统

## file 数据成员

### file Object

```
struct file{  
    unsigned int f_flags;  
    mode_t      f_mode;  
    loff_t      f_pos;  
    ....  
}
```

file\_operations

- file 数据成员并不像 super block和inode 一样，在磁盘上有着对应的结构。
- file结构中并没有dirty标志。

# VFS文件系统

## file 函数成员

### file Object

#### file\_block

```
struct file_operations {  
    llseek(file, offset, origin)  
    read(file, buf, count, offset)  
    write(file, buf, count, offset)  
    readdir(dir, dirent, filldir)  
    poll(file, poll_table)  
    open(inode, file)  
    flush(file)      ....  
}
```

- 每个文件系统都有自己对文件的操作的具体实现。当内核将磁盘上的inode读入内存当中时，对其操作的具体函数地址保存在file\_operations结构中，同时file\_operations结构的地址保存在struct inode中的i\_fop成员变量中。

# VFS文件系统

## dentry Object (/include/linux/Dcache.h)

### dentry Object

```
struct dentry {  
    struct dentry * d_parent;  
    struct list_head d_hash;  
    struct list_head d_lru;  
    struct list_head d_child;  
    struct list_head d_subdirs;  
    ....  
}  
  
struct dentry_operations {  
    ....  
}
```

- 当目录被VFS读入内存中后，便被看作一个dentry对象保存到dentry结构中。
- 对一个进程所要寻找的一个路径来说，内核为其每一级目录创建一个dentry对象。如：

对于 /tmp/test

创建 / tmp test 三个dentry

# VFS文件系统

## dentry Object数据成员

### dentry Object

```
struct dentry {  
    atomic_t d_count;  
    struct inode * d_inode;  
    ....  
}
```

```
struct dentry_operations {  
    ....  
}
```

- dentry对象计数
- 与文件名对应的inode

# VFS文件系统

## dentry Object的状态

---

	状态描述	d_count	d_inode	内存情况
free	没有可用信息	/	/	由调度程序掌管
Unused	dentry当前不被使用	0	指向相应inode	随时可能被回收
In use	内核正在使用	>0	指向相应inode	不会被回收
Negative	相应inode不存在	/	NULL	/

# VFS文件系统

## dentry & inode

---

- 为什么同样描述文件，要使用dentry与inode两套结构？

因为他们各自对文件的描述角度不同。

一个文件可能有好几个文件名，而通过不同的文件名来访问同一文件时权限可能是不同的。

dentry结构代表了逻辑意义上的文件，记录着其逻辑意义上的属性。

inode结构代表物理意义上的文件，记录物理上的属性。

dentry与inode之间是一种“多对一”的关系

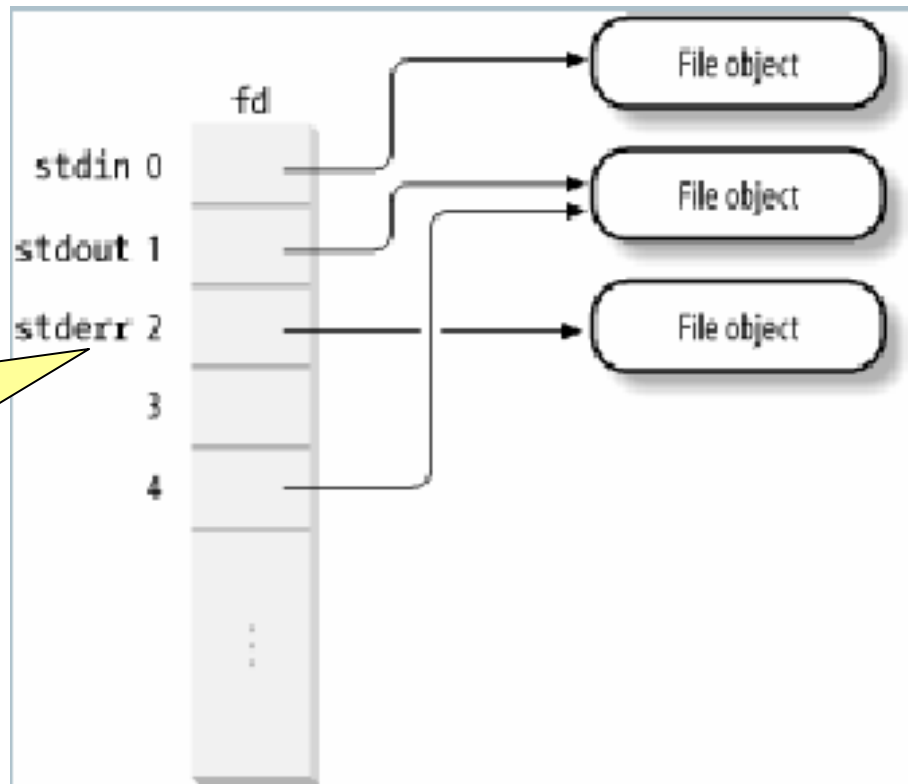


# VFS文件系统

## 进程与文件的关联

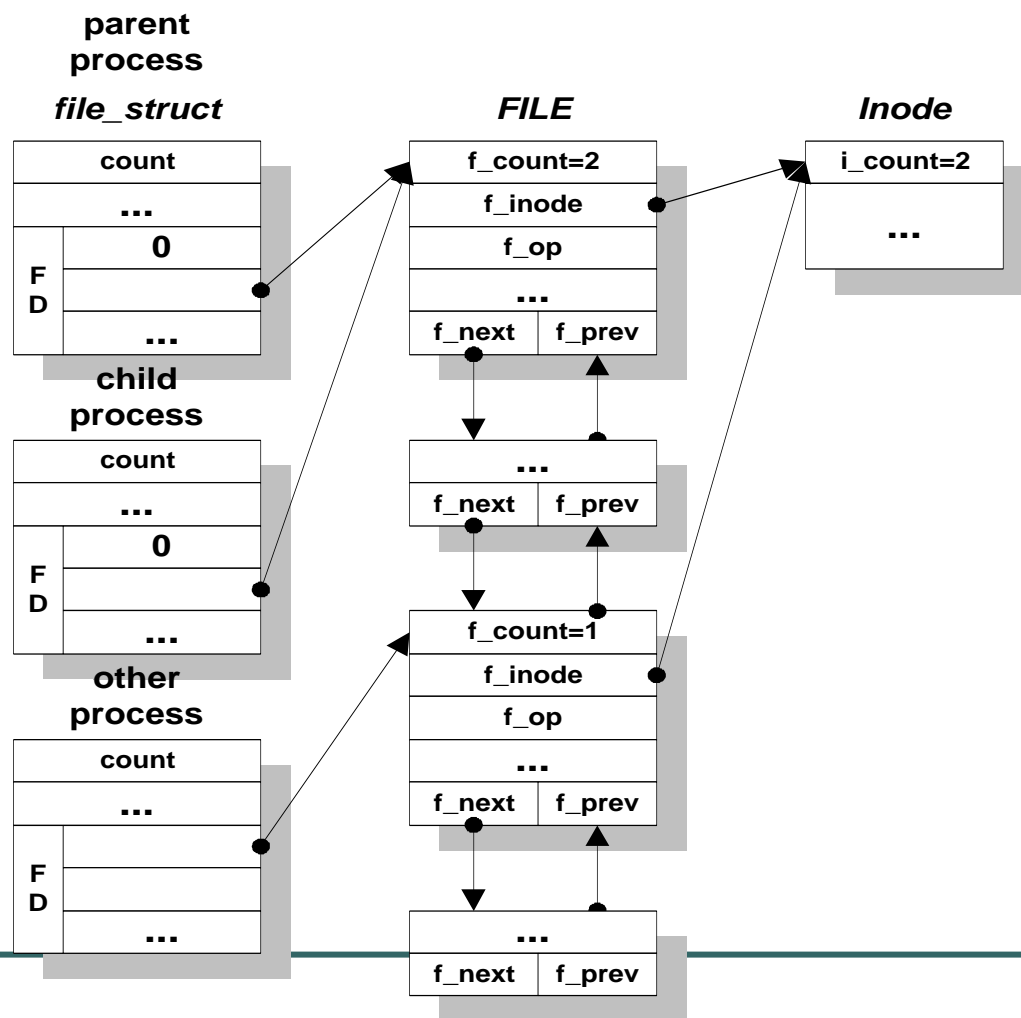
```
struct task_struct {  
    /* open file information */  
    struct files_struct *files;  
}
```

```
struct files_struct {  
    struct file ** fd;  
}
```



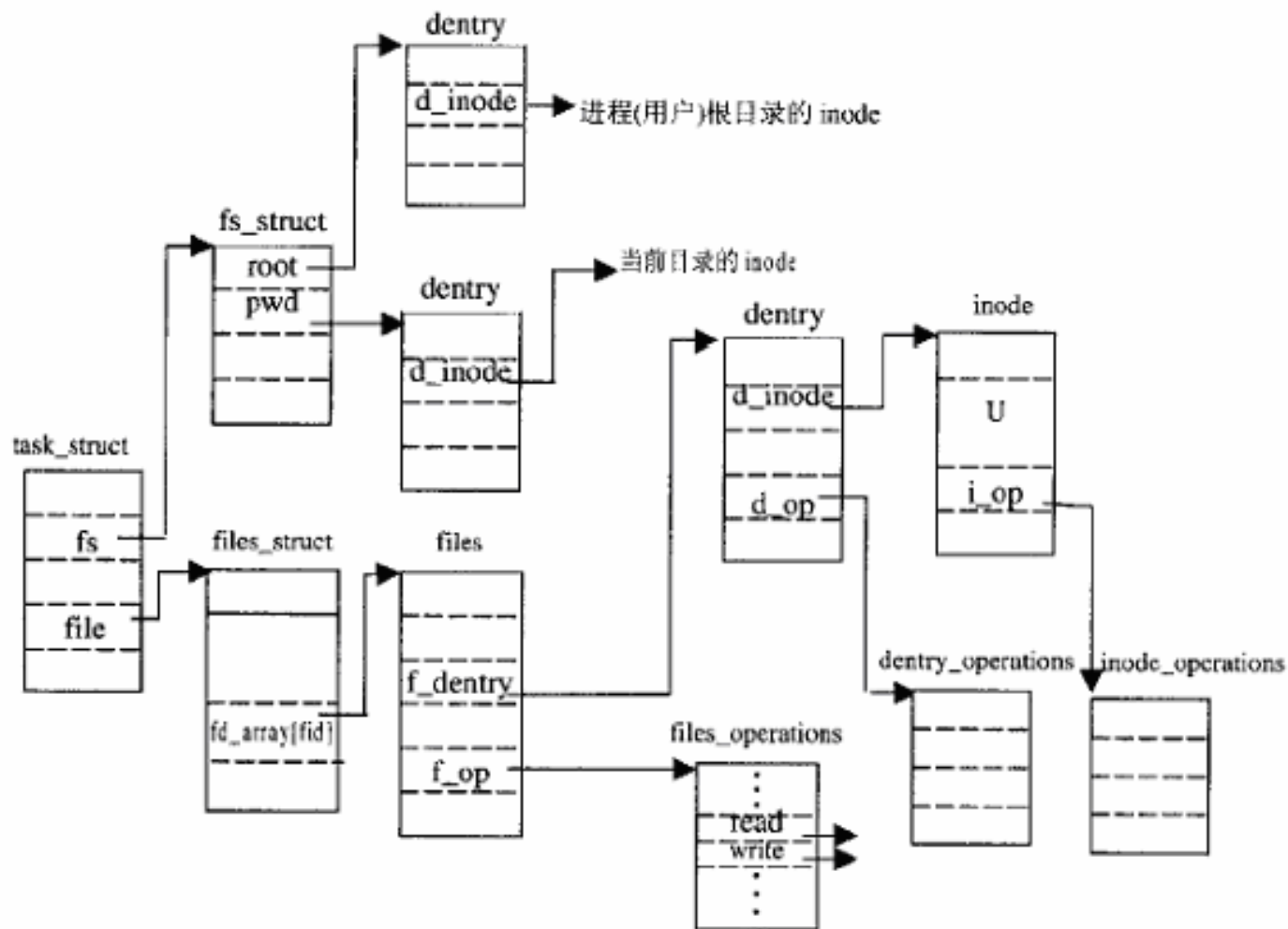
# VFS文件系统

## 进程与文件的关联-续



# VFS文件系统

## 全局逻辑结构



# 文件系统操作

## mount root filesystem

# 文件系统操作

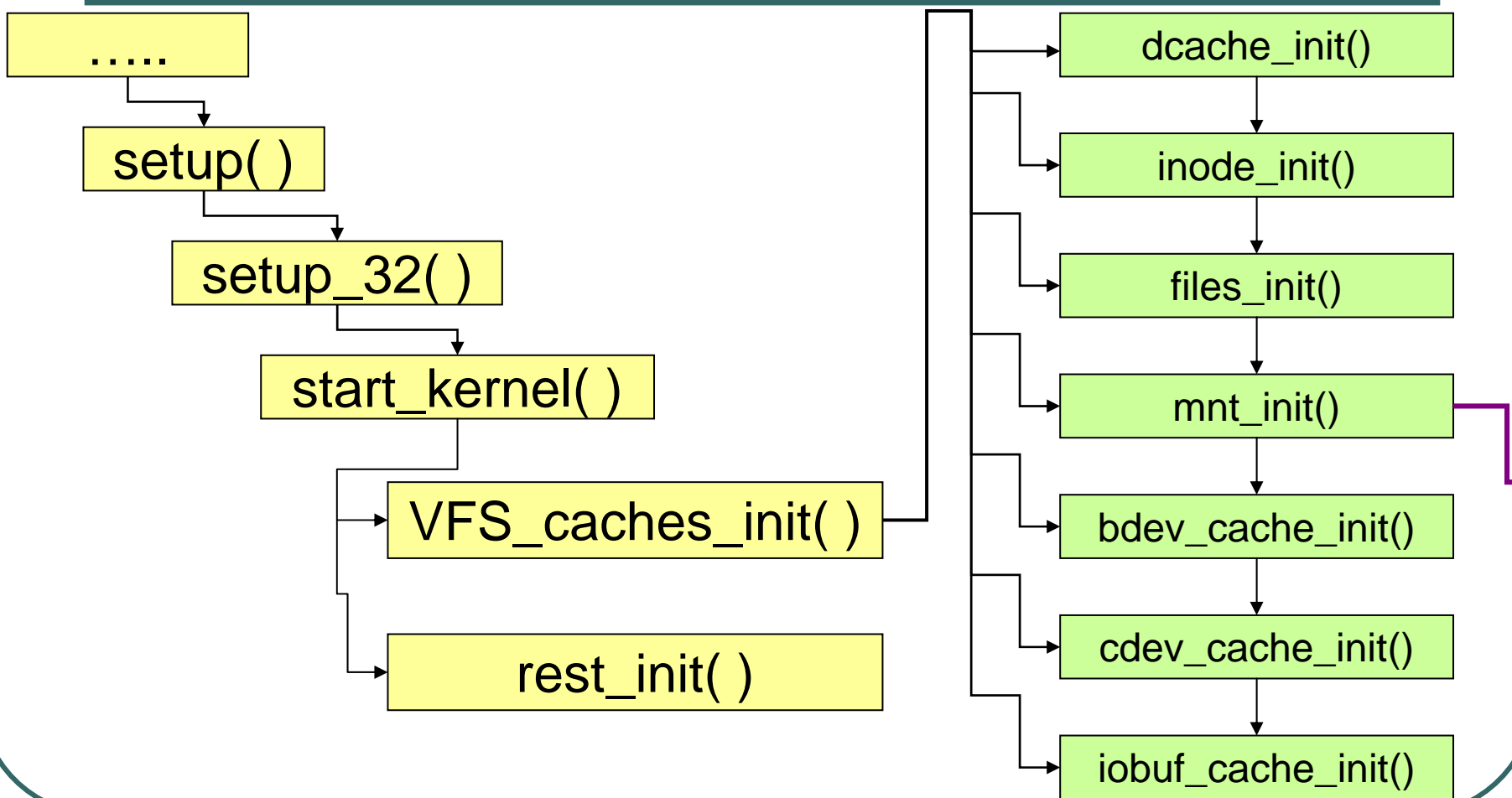
## mount root filesystem

---

- 第一阶段
  - 内核挂载一个特殊的 rootfs 文件系统，它仅是一个空的目录，将作为最初的挂载点
- 第二阶段
  - 内核将真正的root挂载到此空目录上

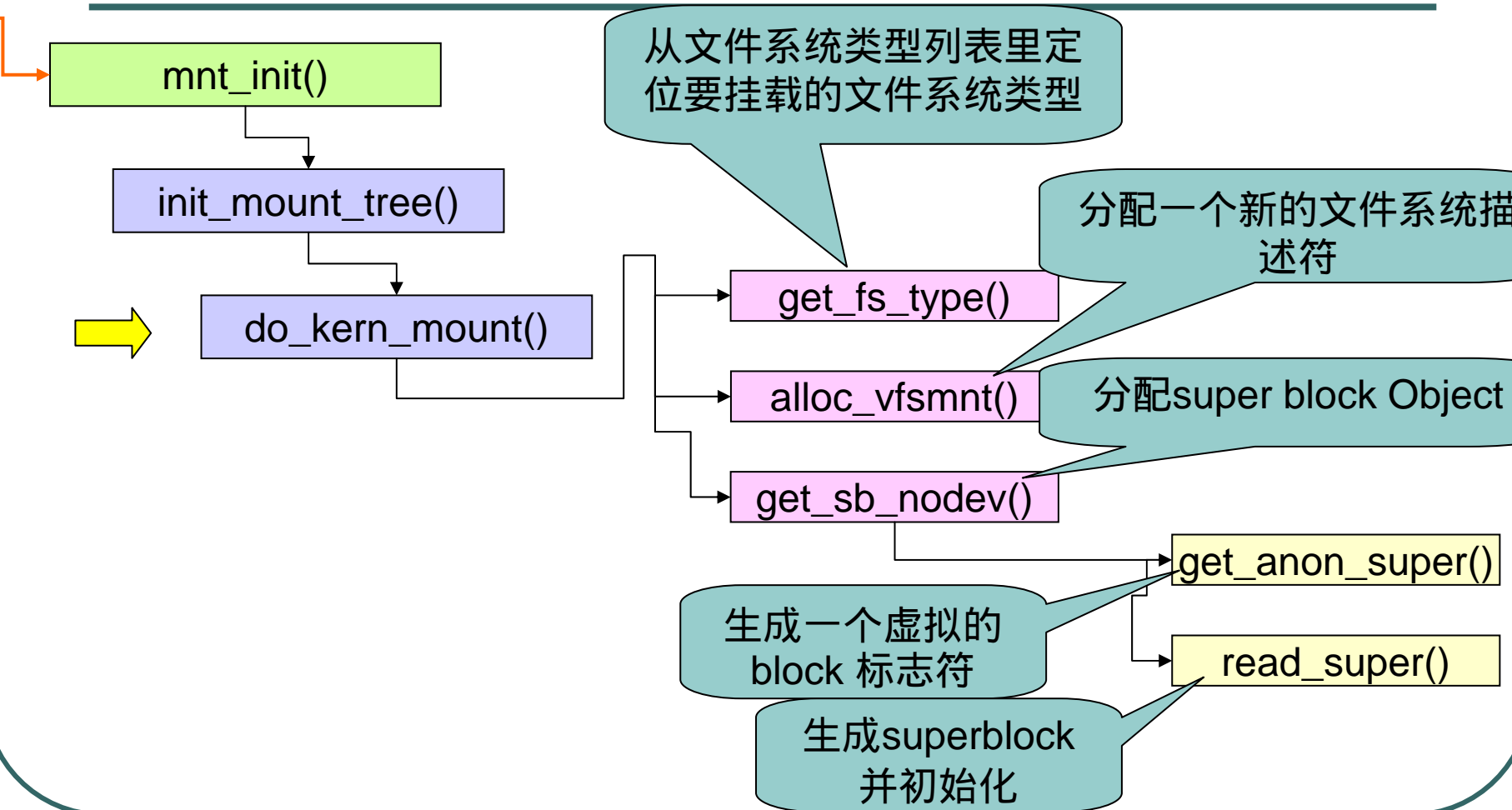
# 文件系统操作mount root filesystem

## 第一阶段-1



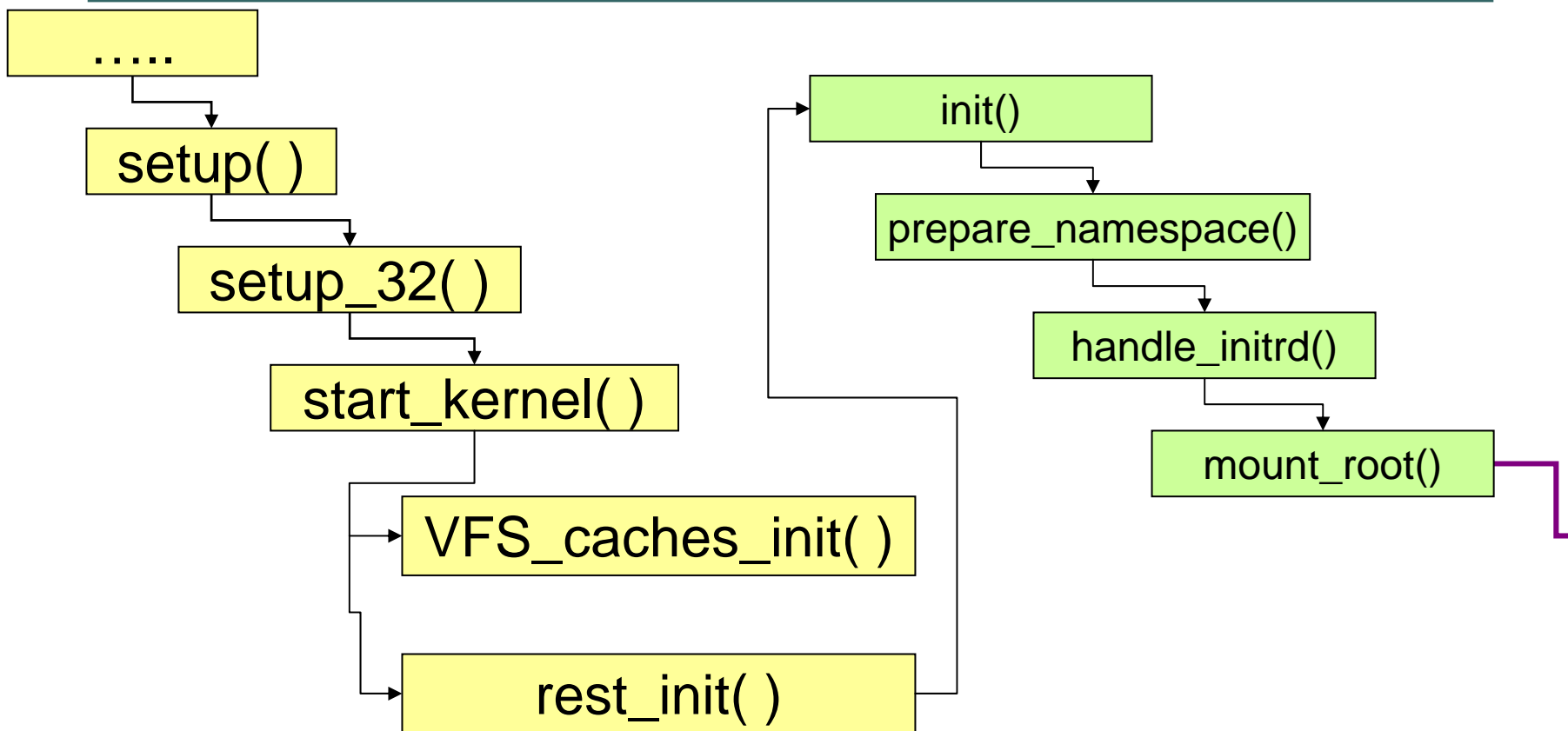
# 文件系统操作mount root filesystem

## 第一阶段-2



# 文件系统操作mount root filesystem

## 第二阶段-1

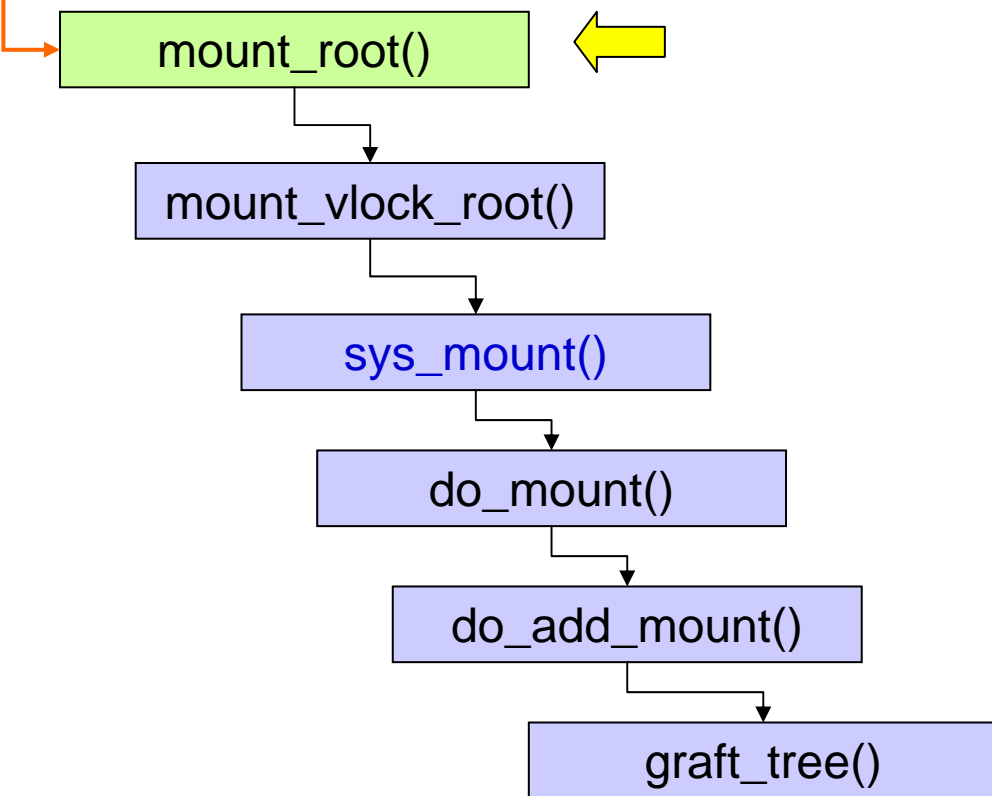




# 文件系统操作mount root filesystem

## 第二阶段-2

---



# 参考文献

---

- Remy Card, Theodore Ts'o, Stephen Tweedie, [Design and Implementation of the Second Extended Filesystem](#)
- Daniel P. Bovet, Marco Cesati , 《Understanding the Linux Kernel, 2nd Edition》
- 李善平 郑扣根 《Linux操作系统及实验教程》
- 毛德操 胡希明 《Linux内核源代码情景分析》
- Linux kernel 2.4.20 源码
- [www.linuxforum.net](http://www.linuxforum.net)

# 谢谢！

敬请关注：

*Linux* 源码分析系列之

## 设备驱动

主讲：杨少华