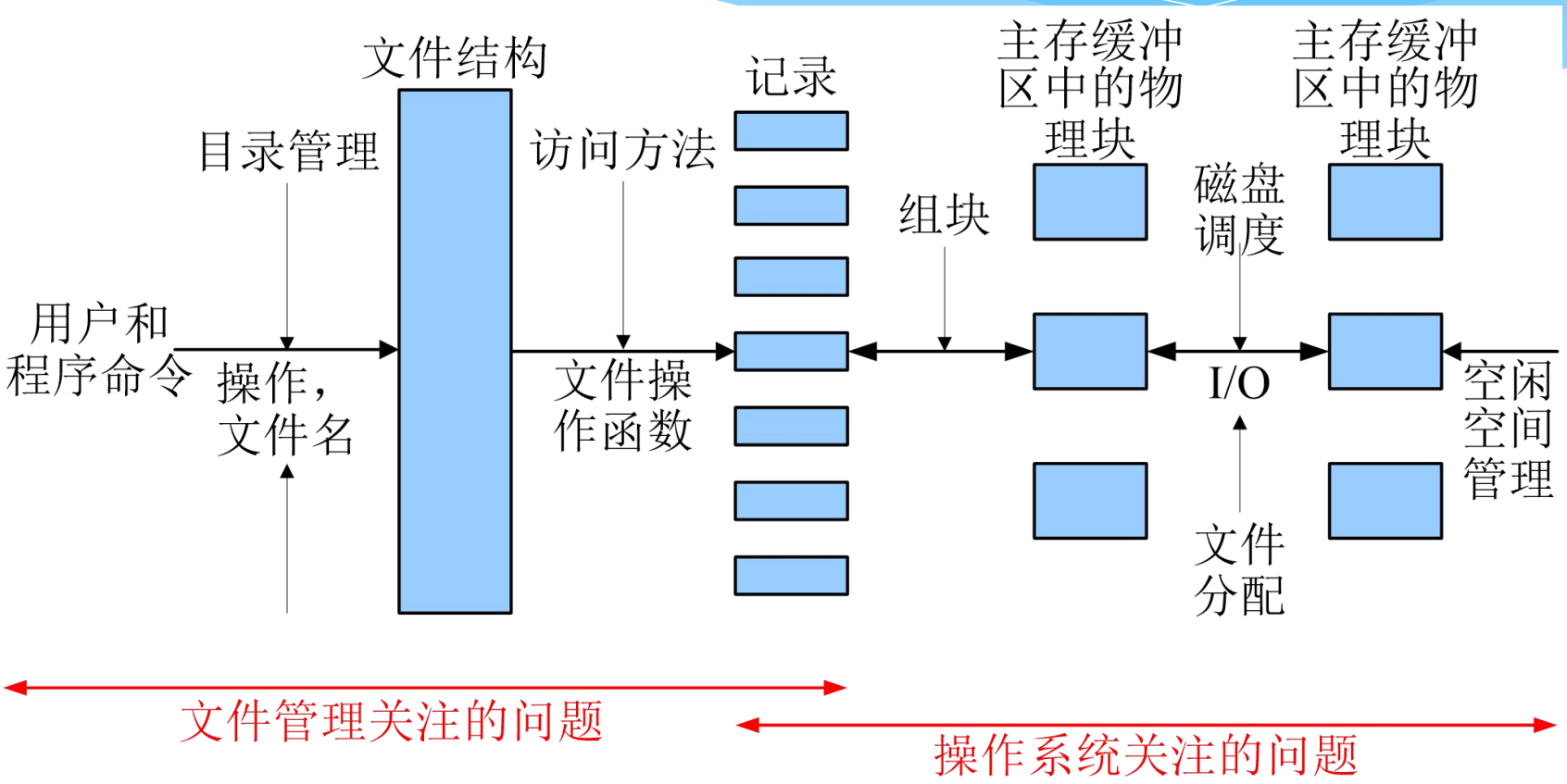
南京大学
NANJING UNIVERSITY

# 计算机操作系统
# 第五章 文件管理(补充)

南京大学软件学院

# 文件管理的要素


文件结构
记录
主存缓冲区中的物理块
主存缓冲区中的物理块
目录管理
访问方法
组块
磁盘调度
用户和程序命令
操作，文件名
文件操作函数
I/O
空闲空间管理
文件分配
文件管理关注的问题
操作系统关注的问题

# 补充内容

1. Inode、目录项、层次目录结构
2. Unix/Linux文件系统的多重索引结构
3. 文件系统的功能与实现

# 1. Inode、目录项、层次目录结构

# Linux特殊目录项建立方法(1)

Linux系统的FCB 中的文件名和其他管理信息分开，其他信息单独组成一个数据结构，称为索引节点 inode，此索引节点在磁盘上的位置由 inode 号标识。

| 文件名 | inode号 |
| --- | --- |
| 最长256个字节 | 4个字节 |

# Inode(1)

* 文件系统中的每个文件都有一个磁盘 inode 与之对应，这些 inode 被集中存放于磁盘上的 inode 区。FCB 对于文件的作用，犹如 PCB 对于进程的作用，集中这个文件的所有相关信息，找到 inode，就能获得此文件的必要信息。

```
* struct  inode {
*    unsigned long i_ino;        /*inode号*/
     atomic_t i_count;           /*inode引用数*/
*    kdev_t i_dev;               /*inode所在设备*/
*    …
*   union {
*   }
};
```

# Inode(2)

* 数据块索引在union结构的每个具体文件系统中，其中的i_data[15]数组给出数据块地址索引，前12项为直接索引，第13项为一次间接索引，第14项为二次间接索引，第15项为三次间接索引。
* 磁盘inode记录文件的属性和相关信息，文件访问过程中会频繁地用到它，不断来回于内外存之间引用它，当然是极不经济的。为此，为此，在内存区开辟一张活动inode表。磁盘inode反映文件静态特性，活动inode反映文件动态特性。

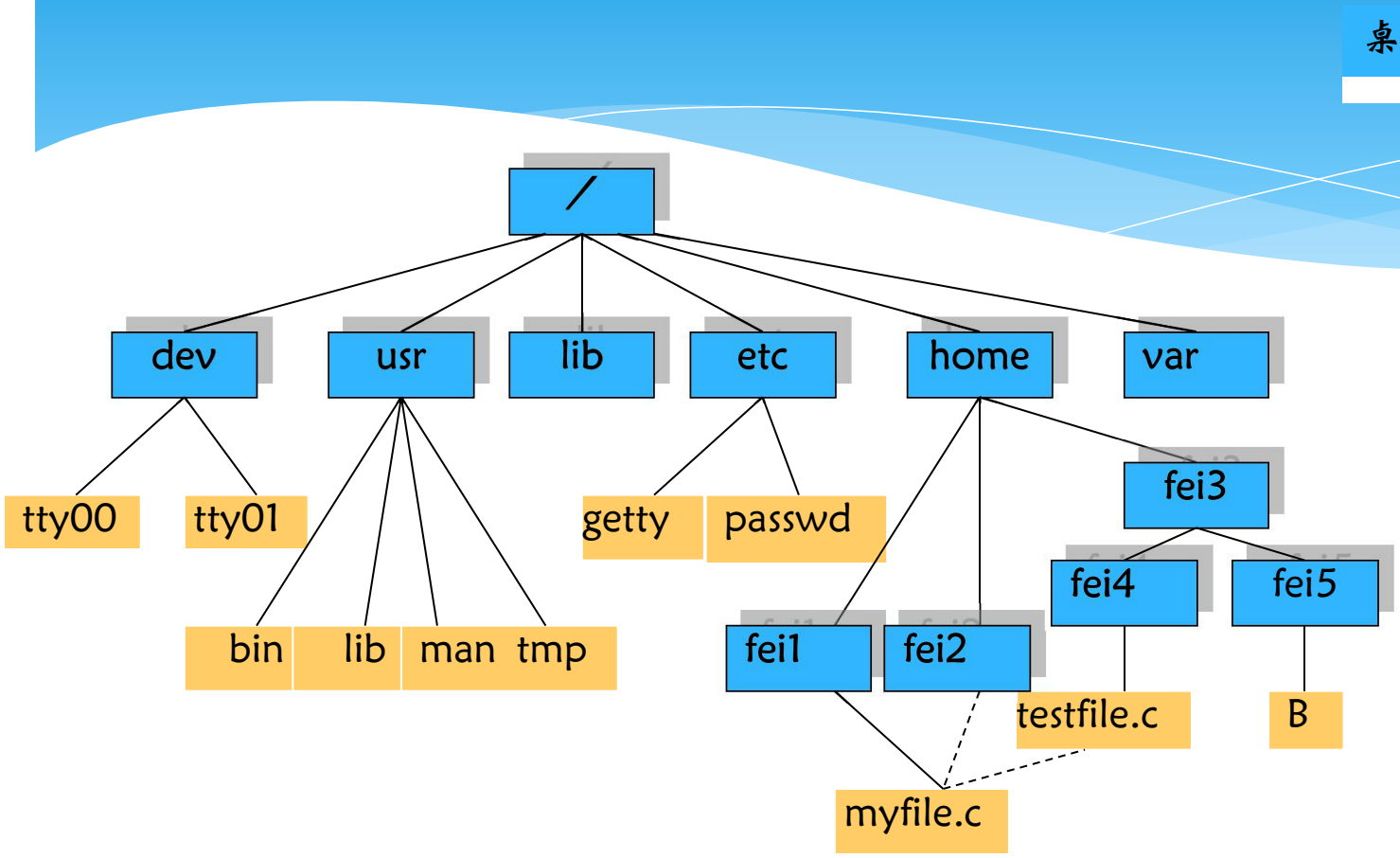
# Inode(3)

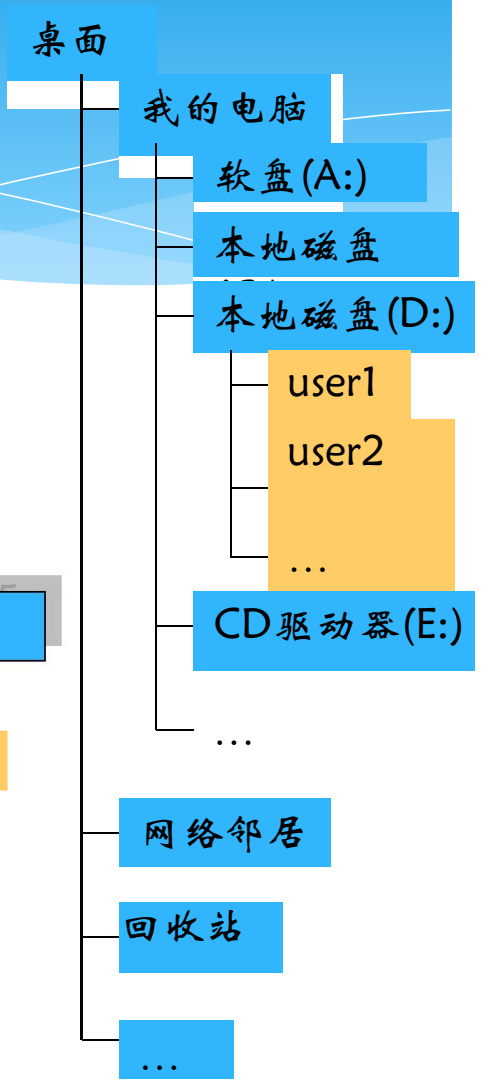
* 当访问某文件时，若在活动 inode 表中找不到其 inode，就申请一个空闲活动 inode，把磁盘 inode 内容复制给它，随之就可用来控制文件读写。
* 当用户关闭文件时，活动 inode 的内容回写到对应的磁盘 inode 中，再释放活动 inode 以供它用。把 FCB 的主要内容与索引节点号分开，不仅能够加快目录检索速度，而且，便于实现文件共享。

# 与文件有关的概念小结

- FCB。
- 文件目录。
- 文件目录项
- 当前目录项“.”和父目录项“..”
- 目录文件。
- inode。
- 静态inode和动态inode。

# 层次目录结构(1)




Linux和Windows层次目录结构

# 层次目录结构(2)

* 路径名
* 绝对路径名
* 相对路径名

# 层次目录结构(3)

* 每一级目录可以是下一级目录的说明，也可以是文件的说明，形成层次关系。
* 多级目录结构采用树型结构，是一棵倒向有根树，树根是根目录；从根向下，每个树枝是一个子目录；而树叶是文件。
* 树型多级目录优点：
  * 较好地反映现实世界中具有层次关系的数据集合和确切地反映系统内部文件的分支结构；
  * 不同文件可重名，只要它们不位于同一末端子目录中，易于规定不同层次或子目录中文件的不同存取权限，便于文件的保护、保密和共享等，有利于系统的维护和查找。

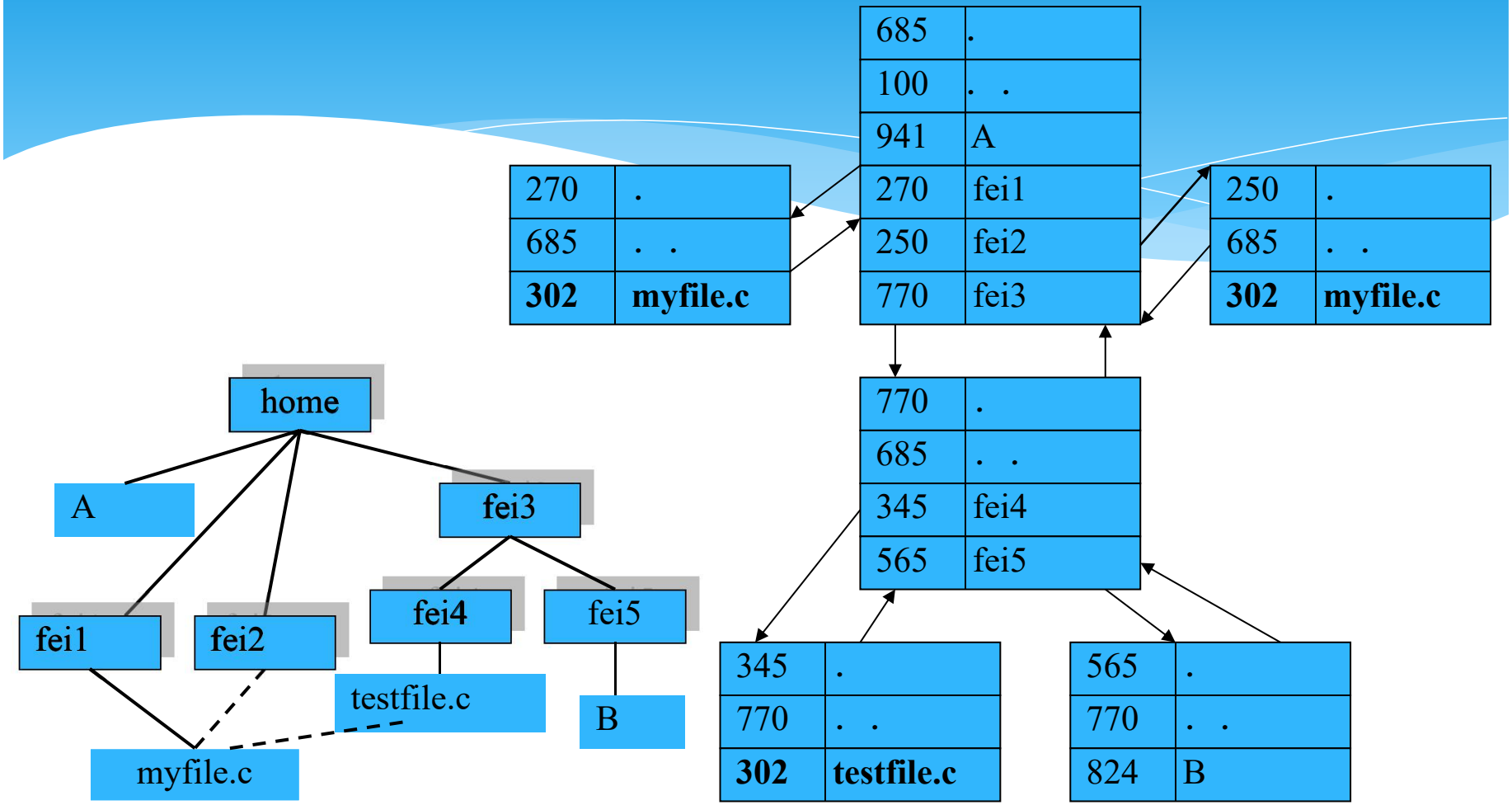
# 层次目录结构(4)

* 如果规定每个文件都只有一个父目录，称为纯树型目录结构，其缺点是文件共享不是对称的，父目录有效拥有该文件，其他被授权用户必须经过属主目录才能对该文件进行访问。
* 有向无环图目录尽管它允许文件有多个父目录而破坏树的特性，但不同用户可以对称方式实现文件共享，即可能属于不同用户的多个目录，使用不同文件名能访问和共享同一个文件。
* 有向无环图目录结构的维护比纯树型目录结构复杂，由于一个文件可能有多个父目录，需为每个文件维护一个引用计数，用来记录文件的父目录个数，仅当引用计数为1时，删除操作才移去文件，否则仅仅把相关记录从父目录中删去。

# 层次目录结构(5)

* Linux支持多父目录，但其中一个是主父目录，它是文件拥有者，且文件被物理存储在此目录下，其他次父目录通过link方式来连结和引用文件，允许任一父目录删除共享文件。
  * 图6-1(a)中便示例这种文件共享的情形，文件/home/fei1为myfile.c的主父目录(图中实线表示)，/home/fei2和/home/fei3/fei4均为文件myfile.c的次父目录(图中虚线表示)。
* Windows实现被称作“快捷方式”的多父目录连结，快捷方式是一些指向不同文件夹(子目录)和菜单之间任意复制和移动的文件及文件夹的指针，删除快捷方式就是删除指针。

# 文件目录的检索




(a)用户角度目录结构

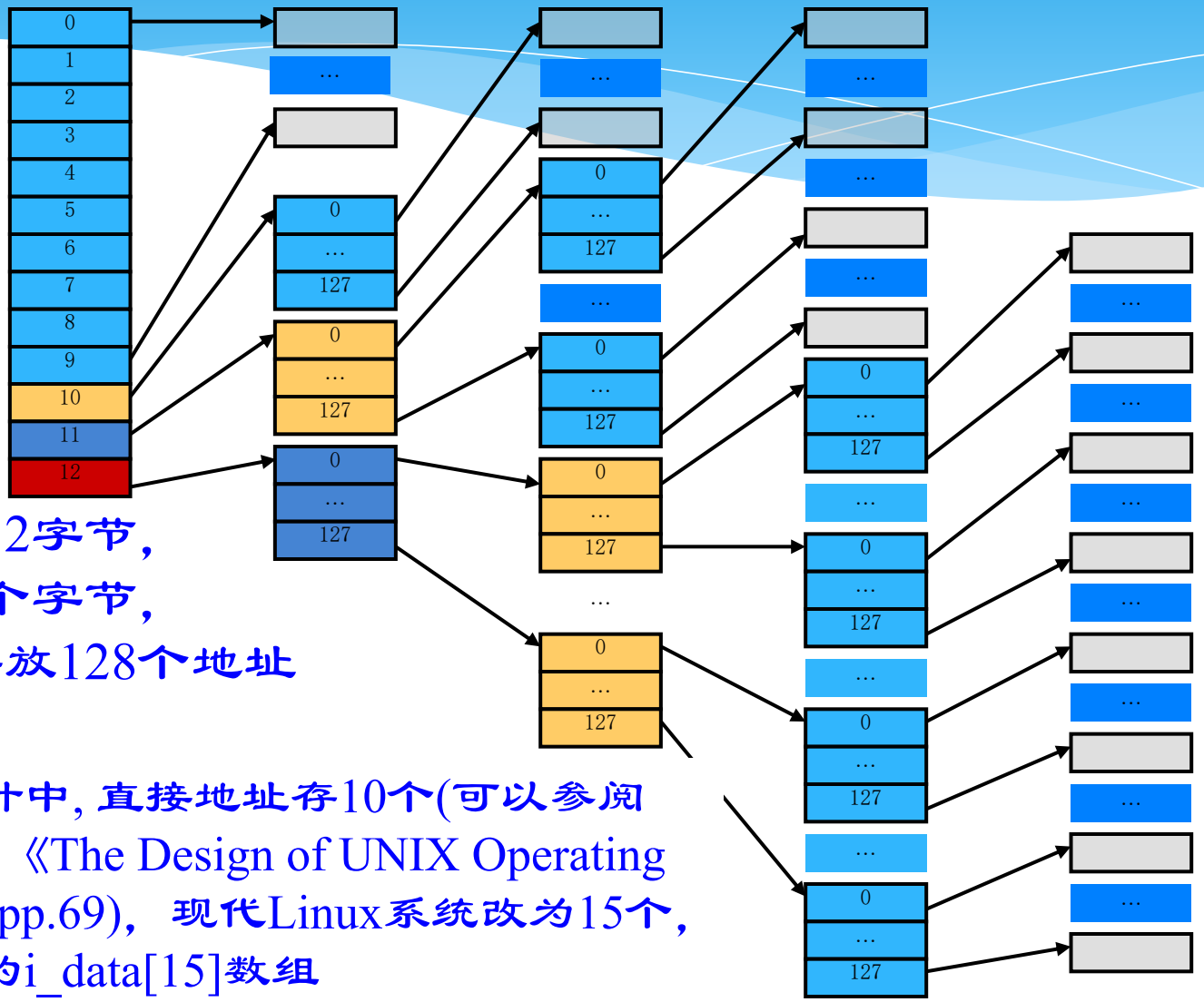
(b)系统角度目录链接

不同角度的目录结构

# 2. Unix/Linux文件系统的多重索引结构

# 索引文件

## UNIX/Linux多重索引结构




每个物理块512字节，

每个块地址4个字节，

每个块可以存放128个地址

Unix最初的设计中，直接地址存10个(可以参阅Maurice J.Bach 《The Design of UNIX Operating System, 1986》pp.69)，现代Linux系统改为15个，即后面会提到的i_data[15]数组

# 索引文件
# UNIX/Linux多重索引结构

* 在UNIX系统中，每个i节点中分别含有10个直接地址的索引和一、二、三级间接索引。若每个盘块放128个盘块地址，则一个1MB的文件分别占用多少各级索引所使用的数据物理块？20MB的文件呢？设每个盘块有512B。
* 答：

直接块容量=10×512B/1024=5KB

一次间接容量=128×512B/1024=64KB

} 69KB

二次间接容量=128×128×512B/1024=64KB×128=8192KB

三次间接容量
=128×128×128×512B/1024=64KB×128=8192KB×128=1048576KB

1MB为1024KB，1024KB-69KB=955KB，955×1024B/512B=1910块，1MB的文件分别占用1910个二次间接盘块，128个一次间接盘块。

20×1024KB-69-8192=12219KB，12219×1024B/512=24438块，20MB的文件分别占用24438个三次间接盘块和128*128=16384个二次间接盘块，128个一次间接盘块。

# 3. 文件系统的功能与实现

# 3.文件系统的功能与实现

3.1 文件系统调用的实现

3.2 文件共享

3.3 文件空间管理

3.4 主存映射文件

3.5 虚拟文件系统

# 3.1 文件操作的实现

* 文件系统提供给用户程序的一组系统调用，包括：创建、删除、打开、关闭、读、写和控制，通过这些系统调用用户能获得文件系统的各种服务。
* 在为应用程序服务时，文件系统需要沿路径查找目录以获得该文件的各种信息，这往往要多次访问文件存储器，使访问速度减慢，若把所有文件目录都复制到主存，访问速度可加快，但却又增加主存开销。

# 3.1 文件操作的实现

* 方案：把常用和正在使用的那些文件目录复制进主存，这样，既不增加太多主存开销，又可明显减少查找时间，
* 系统为每个用户进程建立一张活动文件表，用户使用文件之前先通过“打开”操作，把该文件的文件目录复制到指定主存区域，
* 当不再使用该文件时，使用“关闭”操作切断和该文件目录的联系，这样，文件被打开后，可被用户多次使用，直至文件被关闭或撤销，大大减少访盘次数，提高文件系统的效率。

# 文件系统磁盘结构

1 超级块：占用1#号块
2 索引节点区：2#～k+1#块
3 数据区：k+2#～n#为数据块

# 文件系统磁盘结构

1 超级块：占用1#号块

占用1#号块，存放文件系统结构和管理信息，

如记录inode表所占盘块数、

文件数据所占盘块数、

主存中登记的空闲盘块数、

主存中登记的空闲块物理块号、

主存中登记的空闲inode数、

主存中登记的空闲inode编号，

及其他文件管理控制信息，

可见超级块既有盘位示图的功能，又记录整个文件卷的控制数据。

每当一个块设备作为文件卷被安装时，该设备的超级块就要复制到主存系统区中备用，而拆卸文件卷时，修改过的超级块需复制回磁盘的超级块中。

# 文件系统磁盘结构

1 超级块：占用1#号块

2 索引节点区：2#～k+1#块

存放inode表，每个文件都有各种属性，它们被记录在称为索引节点inode的结构中；所有inode都有相同大小，且inode表是inode结构的列表，文件系统中的每个文件在该表中都有一个inode。又分磁盘inode表和主存活动inode表，后者解决频繁访问磁盘inode表的效率问题。

3 数据区：k+2#～n#为数据块

# 文件系统磁盘结构

1 超级块：占用1#号块

2 索引节点区：2#～k+1#块

3 数据区：k+2#～n#为数据块

文件的内容保存在这个区域，磁盘上所有物理块的大小是一样的，如果文件包含超过一块的数据，则文件内容会存放在多个盘块中。

# 文件系统磁盘结构

重要数据结构：

1 用户打开文件表：

进程的PCB结构中保留一个files_struct，称为用户打开文件表或文件描述符表，表项的序号为文件描述符fd，该登记项内登记系统打开文件表的一个入口指针fp，通过此系统打开文件表项连接到打开文件的活动inode。

2 系统打开文件表：

# 文件系统磁盘结构

重要数据结构：

1 用户打开文件表：

2 系统打开文件表：

是为解决多用户进程共享文件、父子进程共享文件而设置的系统数据结构file_struct，主存专门开辟最多可登记256项的系统打开文件表区，当打开一个文件时，通过此表项把用户打开文件表的表项与文件活动inode联接起来，以实现数据的访问和信息的共享。

3 主存活动inode表：

# 文件系统磁盘结构

重要数据结构：

1 用户打开文件表：

2 系统打开文件表：

3 主存活动inode表：

为解决频繁访问磁盘索引节点inode表的效率问题，系统开辟的主存区，正在使用的文件的inode被调入主存活动索引节点inode中，以加快文件访问速度。

# 文件系统内部结构

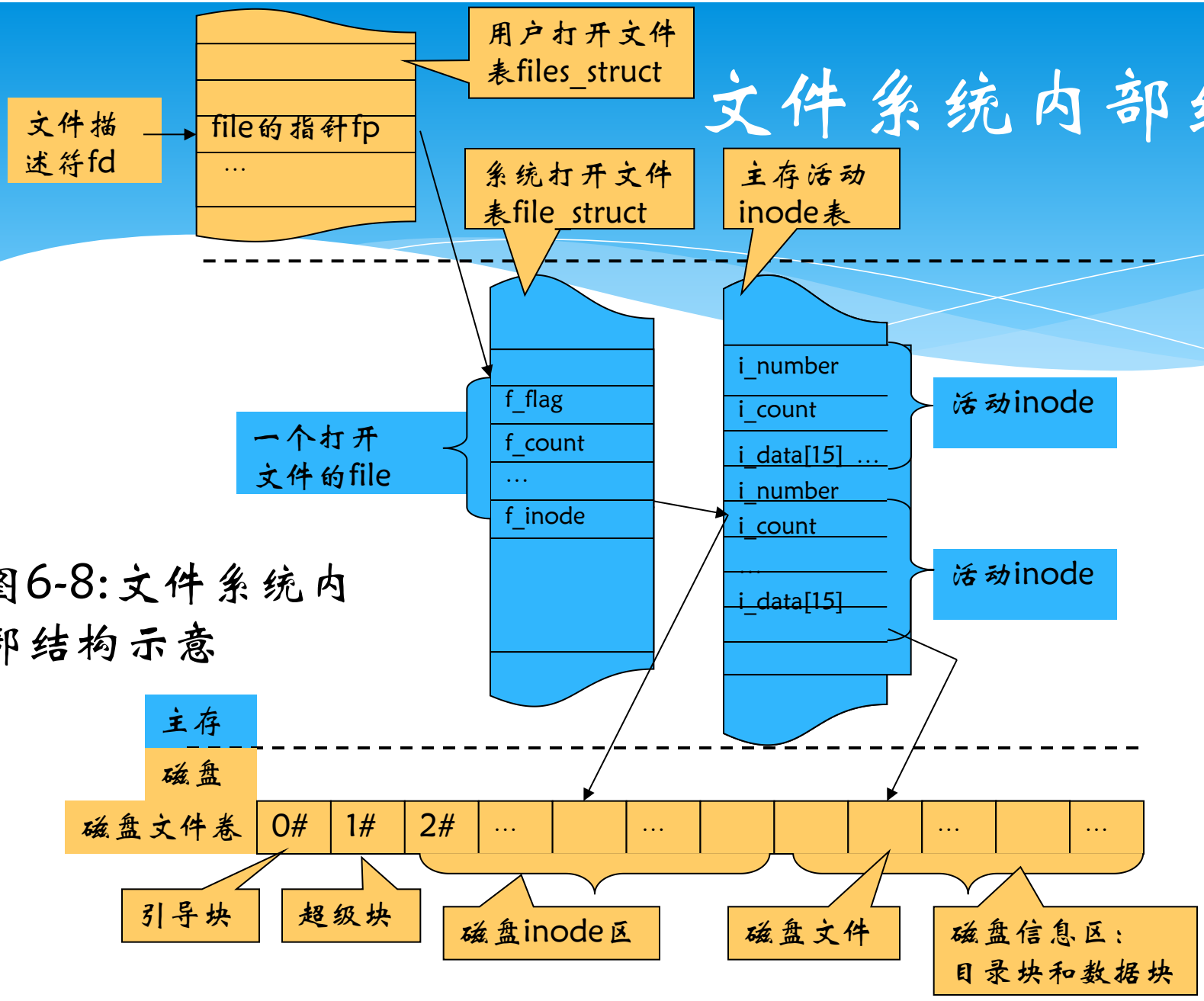
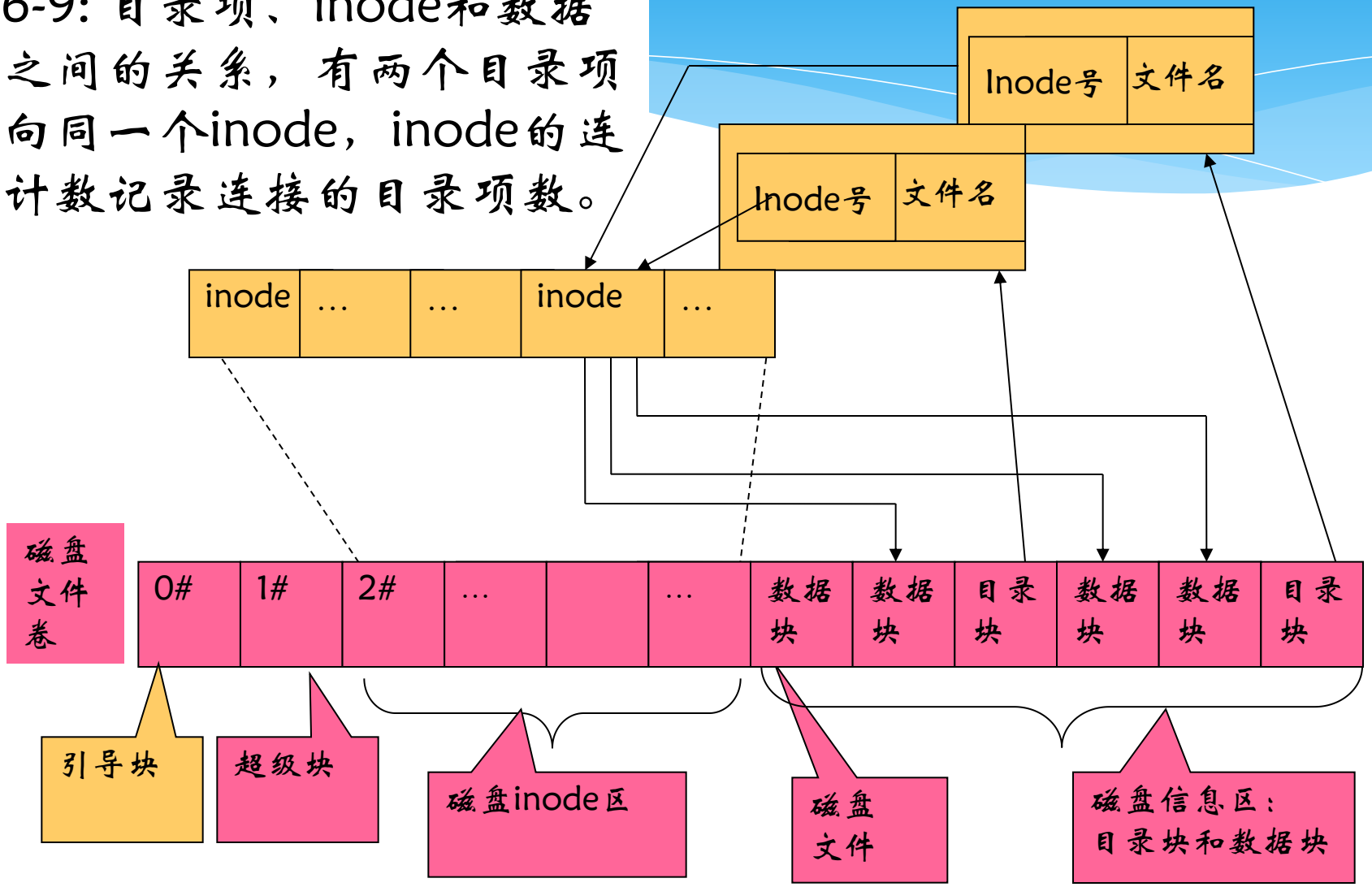



图6-8:文件系统内部结构示意

# 目录项、inode和数据块的关系

图6-9：目录项、inode和数据块之间的关系，有两个目录项指向同一个inode，inode的连接计数记录连接的目录项数。



# 文件系统调用 (1)
# (1)文件的创建

系统调用C语言格式为：

```
int fd, mode;
char *filenamep;
fd = create (filenamep, mode);
```

# 文件系统调用 (2)

## 文件创建执行过程

① 为新文件分配索引节点和活动索引节点，并把索引节点编号与文件分量名组成新目录项，记到目录中。

② 在新文件所对应的活动索引节点中置初值，如置存取权限i_mode，连接计数i_nlink等。

③ 分配用户打开文件表项和系统打开文件表项，置表项初值，读写位移f_offset清“0”。

④把各表项及文件对应的活动索引节点用指针连接起来，把文件描述字返回给调用者。

# 文件系统调用 (3)
## (2) 文件的删除

* 删除把指定文件从所在的目录文件中除去。
* 如果没有连接用户(i_link 为“1”)，还要把文件占用的存储空间释放。删除系统调用形式为：unlink (filenamep)。
* 在执行删除时，必须要求用户对该文件具有“写”操作权。

# 文件系统调用 (4)
## (3) 文件的打开(1)

调用方式为：

```
int fd, mode;
char * filenamep;
fd = open (filenamep, mode);
```

# 文件系统调用 (5)

## 文件打开执行过程(2)

① 检索目录，把它的外存索引节点复制到活动索引节点表。

② 根据参数mode核对权限，如果非法，则这次打开失败。

③ 当“打开”合法时，为文件分配用户打开文件表项和系统打开文件表项，并为表项设置初值。通过指针建立这些表项与活动索引节点间的联系。把文件描述字，即用户打开文件表中相应文件表项的序号返回给调用者。

# 文件系统调用 (6)
## (4)文件的关闭(1)

调用方式为：

```
int fd;
close (fd);
```

# 文件系统调用 (7)
## 文件的关闭(2)

① 根据fd找到用户打开文件表项，再找到系统打开文件表项。释放用户打开文件表项。

② 把对应系统打开文件表项中的f_count减“1”，如果非“0”，说明还有进程共享这一表项，不用释放直接返回；否则释放表项。

③ 把活动索引节点中的i_count减“1”，若不为“0”，表明还有用户进程正在使用该文件，不用释放而直接返回，否则在把该活动索引节点中的内容复制回文件卷上的相应索引节点中后，释放该活动索引节点。

# 文件系统调用 (8)
## 文件的关闭(3)

④ f_count和i_count分别反映进程动态地共享一个文件的两种方式，

a) f_count反映不同进程通过同一个系统打开文件表项共享一个文件的情况；

b) i_count反映不同进程通过不同系统打开文件表项共享一个文件的情况；

c) 通过两种方式，进程之间既可用相同的位移指针f_offset，也可用不同位移指针f_offset共享同一个文件。

# 文件系统调用 (9)
## (5)读文件(1)

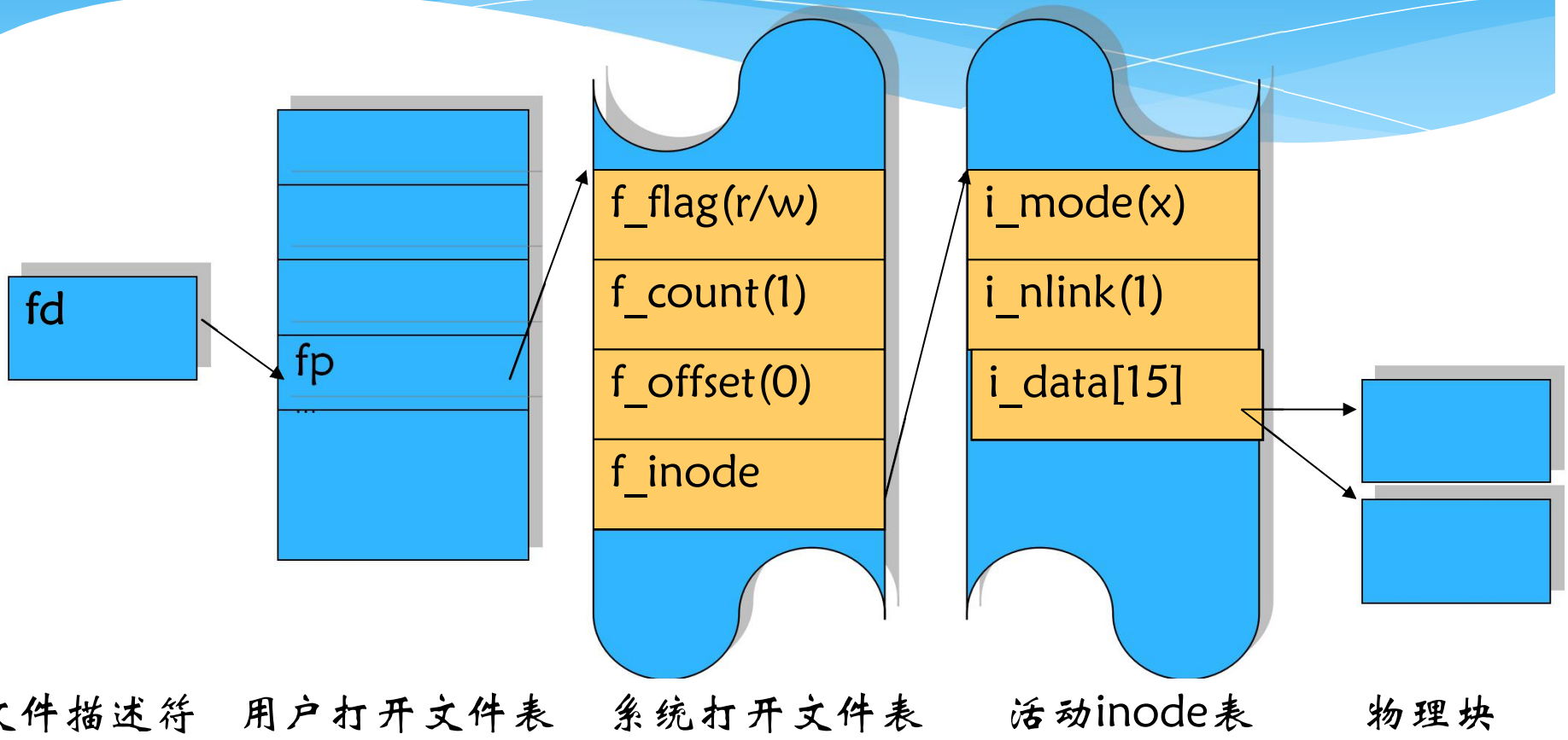
调用的形式为：

```
int nr, fd, count;
char buf [ ]
nr = read (fd, buf, count);
```

# 文件系统调用 (10)
## 读文件(2)

* 系统根据f_flag中的信息，检查读操作合法性，
* 再根据当前位移量f_offset值，要求读出的字节数，及活动索引节点中i_data[15]指出的文件物理块存放地址，把相应的物理块读到缓冲区中，然后再送到bufp指向的用户主存区中。

# 文件系统调用 (11)
## 读文件(3)




读操作时文件数据结构的关系

# 文件系统调用 (12)
# (6) 写文件

调用的形式为：

nw = write (fd, buf, count);

buf是信息传送的源地址，即把buf所指向的用户主存区中的信息，写入到文件中。

# 文件系统调用 (13)
# (7)文件的随机存取(1)

* 在文件初次“打开”时，文件的位移量f_offset清空为零，以后的文件读写操作总是根据offset的当前值，顺序地读写文件。为了支持文件的随机访问，提供系统调用lseek，它允许用户在读、写文件前，事先改变f_offset的指向

系统调用的形式为：

long lseek;

long offset;

int whence, fd;

lseek (fd, offset, whence);

# 文件系统调用 (14)
# 文件的随机存取(2)

文件描述字fd必须指向一个用读或写方式打开的文件，

* 当whence是“0”时，则f_offset被置为offset，
* 当whence是“1”时，则f_offset被置为文件当前位置加上offset。

# 3.2文件共享

1文件的静态共享

2文件的动态共享

3文件的符号链接共享

# 1文件的静态共享(1)

系统调用形式为：

chat * oldnamep, * newnamep;

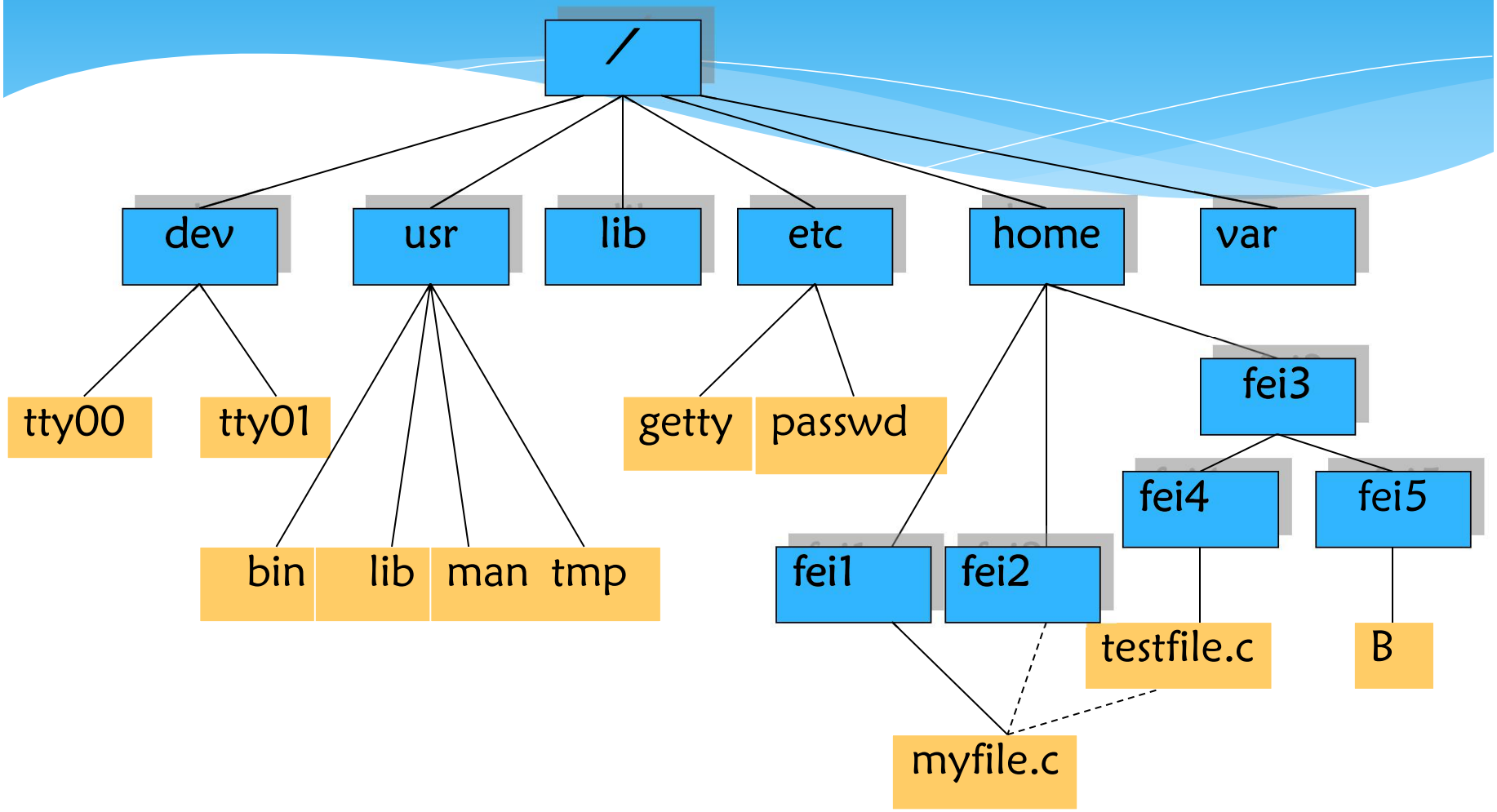
link (oldnamep, newnamep);

① 检索目录找到oldnamep所指向文件的索引节点inode编号。

② 再次检索目录找到newnamep所指文件的父目录文件，并把已存在文件的索引节点inode编号与别名构成一个目录项，记入到该目录中去。

③ 把已存在文件索引节点inode的连接计数i_nlink加“1”。

# Linux层次目录结构



# 文件的静态共享(2)

链接实际上是共享已存在文件的索引节点inode，完成链接的系统调用：　　link(“/home/fei1/myfile.c”,”/home/fei2/myfile.c”)；
link(“/home/fei1/myfile.c”,“/home/fei3/fei4/testfile.c”)；

执行后，三个路径名指的是同一个文件：

/home/fei1/myfile.c，

/home/fei2/myfile.c，

/home/fei3/fei4/testfile.c。

# 文件的静态共享(3)

文件解除链接调用形式为：

unlink (namep)

解除链接与文件删除执行的是同一系统调用代码。删除文件是从文件主角度讲的，解除文件连接是从共享文件的其他用户角度讲的。都要删去目录项，把i_nlink减“1”，不过，只有当i_nlink减为“0”时，才真正删除文件。

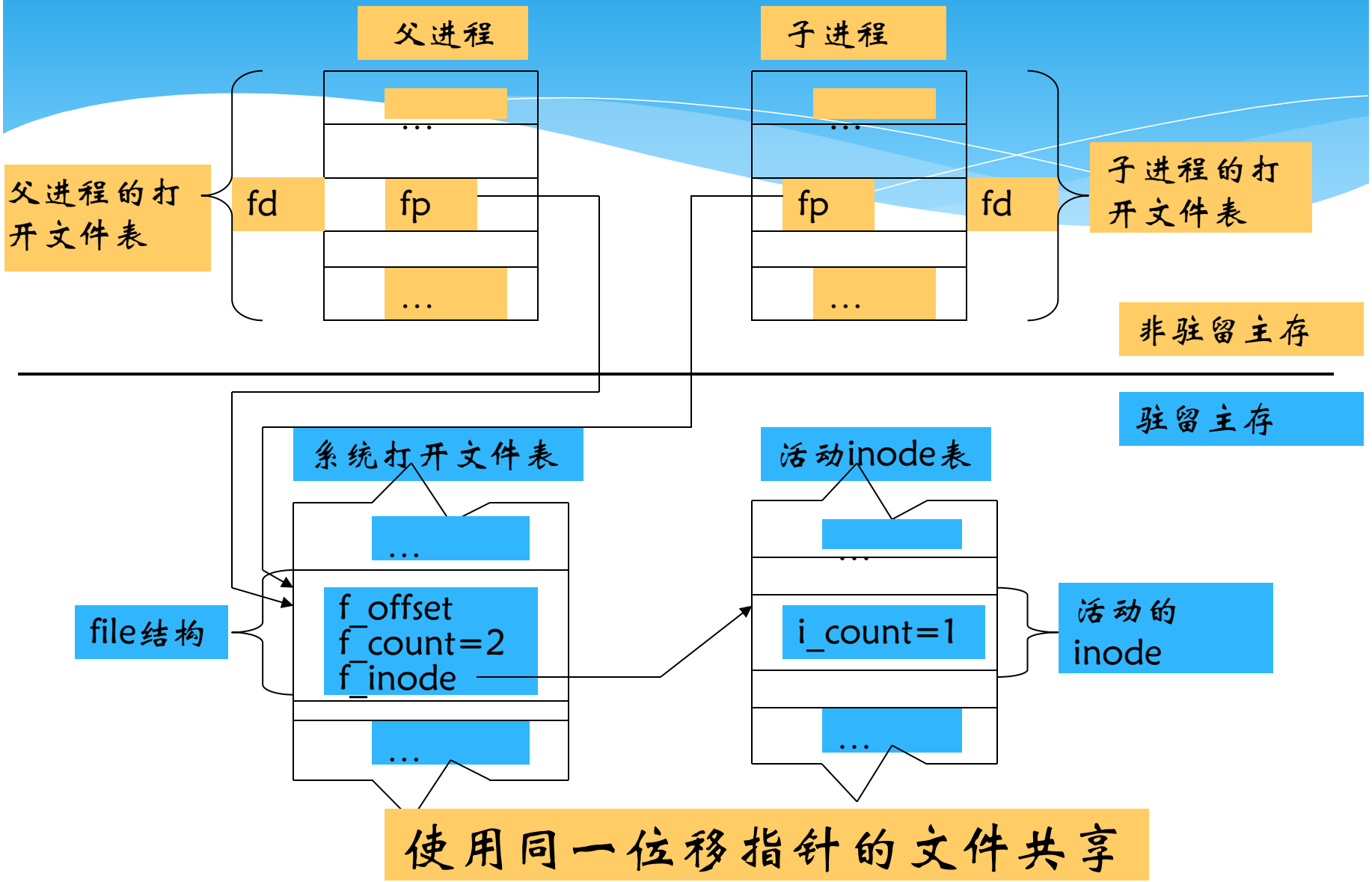
# 2文件的动态共享(1)

* 文件动态共享是系统中不同的用户进程或同一用户的不同进程并发访问同一文件。
* 这种共享关系只有当用户进程存在时才可能出现，一旦用户的进程消亡，其共享关系也就自动消失。
* 文件的每次读写由一个读/写位移指针指出要读写的位置。现在的问题是：应让多个进程共用同一个读/写位移，还是各个进程具有各自的读写位移呢？

# 文件的动态共享(2)

* 同一用户父、子进程协同完成任务，使用同一读/写位移，同步地对文件进行操作。
* 该位移指针宜放在相应文件的活动索引节点中。当用系统调用fork建立子进程时，父进程的PCB结构被复制到子进程的PCB结构中，使两个进程的打开文件表指向同一活动的索引节点，达到共享同一位移指针的目的。

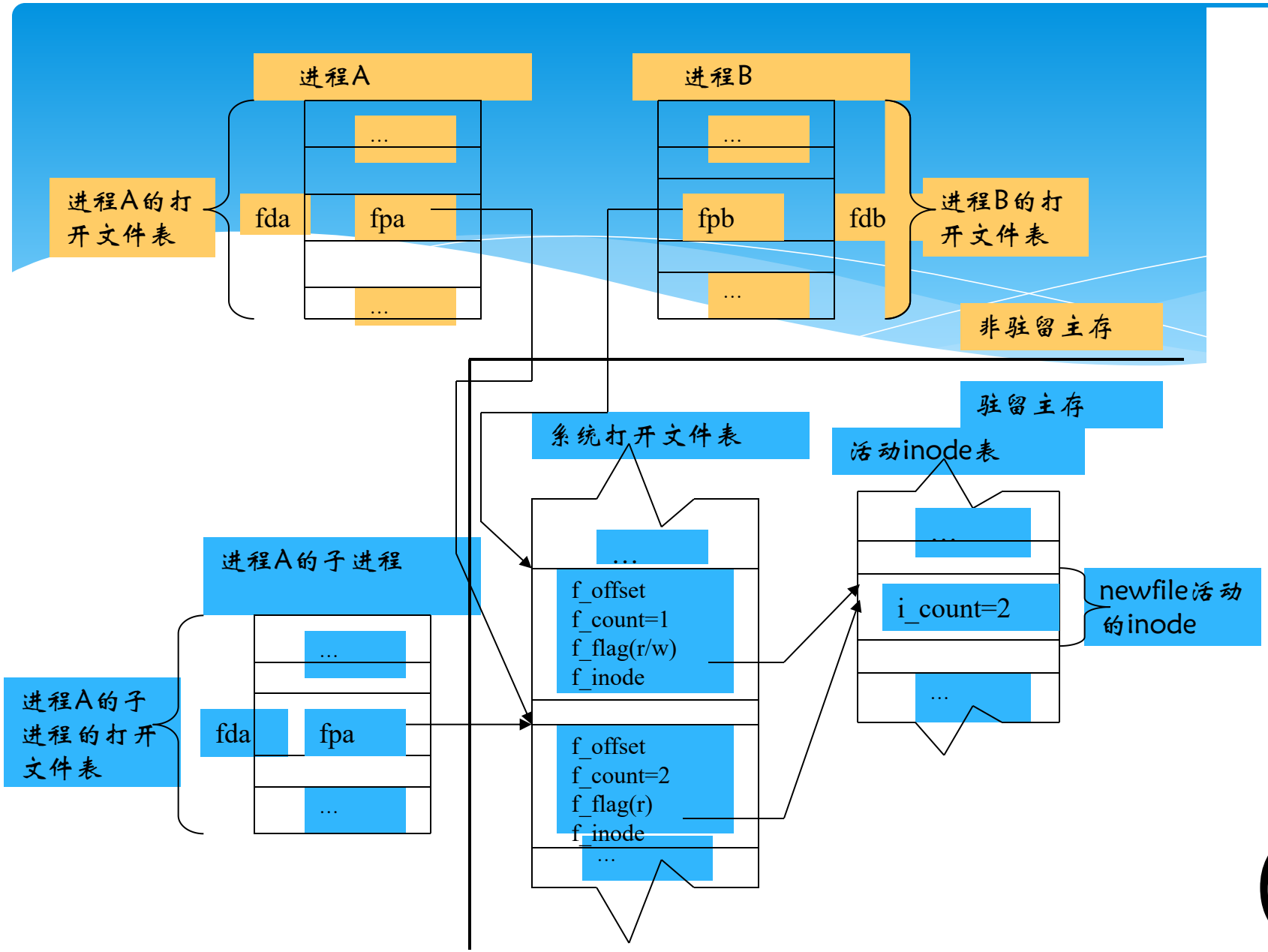
# 文件的动态共享(3)


父进程
子进程
父进程的打开文件表
fd
fp
fp
fd
子进程的打开文件表
非驻留主存
驻留主存
系统打开文件表
活动inode表
file结构
f_offset
f_count=2
f_inode
i_count=1
活动的inode
使用同一位移指针的文件共享

# 文件的动态共享(4)

* 多用户共享文件，每个希望独立地读、写文件，这时不能只设置一个读写位移指针，须为每个用户进程分别设置一个读、写位移指针。
* 位移指针应放在每个进程用户打开文件表的表目中。这样，当一个进程读、写文件，并修改位移指针时，另一个进程的位移指针不会随之改变，从而，使两个进程能独立地访问同一文件。

# 文件的动态共享(5)




使用不同位移指针的文件共享

# 3文件的符号链接共享(1)

* 操作系统可支持多个物理磁盘或多个逻辑磁盘(分区)，那么，文件系统是建立一棵目录树还是多棵目录树呢?
* Windows采用将盘符或卷标分配给磁盘或分区，并将其名字作为文件路径名的一部分。
* UNIX/Linux的每个分区有自己的文件目录树，当有多个文件系统时，可通过安装的办法整合成一棵更大的文件目录树。

# 3文件的符号链接共享(2)

* 问题：系统中每个文件对应一个inode，编号是惟一的，但两个不同的磁盘或分区都含有相同inode号对应的文件，也就是说，整合的目录树中，inode号并不惟一地标识一个文件，
* 办法：拒绝创建跨越文件系统的硬链接。

# 文件的符号链接共享(3)

* 符号链接又称软链接，是一种只有文件名，不指向inode的文件
* 符号链接共享文件的实现思想：

用户A目录中形式为afile→bfile，实现A的目录与B的文件的链接。其中只包含被链接文件bfile的路径名而不是它的inode号，而文件的拥有者才具有指向inode的指针 。

# 文件的符号链接共享(4)

* 当用户A要访问被符号链接的用户B的文件bfile，且要读“符号链接”类文件时，被操作系统截获，它将依据符号链接中的路径名去读文件，于是就能实现用户A使用文件名afile对用户B的文件bfile的共享。
* 优点：能用于链接计算机系统中不同文件系统中的文件，可链接计算机网络中不同机器上的文件，此时，仅需提供文件所在机器地址和该机器中文件的路径名。
* 缺点：搜索文件路径开销大，需要额外的空间查找存储路径。

# 3.3 文件空间管理

磁盘文件空间分配采用两种办法

- 连续分配：文件存放在辅存空间连续存储区中，在建立文件时，用户必须给出文件大小，然后，查找到能满足的连续存储区供使用。
- 非连续分配：一种方法是以块(扇区)为单位，扇区不一定要连续，同一文件的扇区按文件记录的逻辑次序用链指针连接或位示图指示。
- 另一种方法是以簇为单位，簇是由若干个连续扇区组成的分配单位；实质上是连续分配和非连续分配的结合。各个簇可以用链指针、索引表，位示图来管理。

# 磁盘空闲空间管理方法

* 位示图
* 空闲区表
* 空闲块链

# 磁盘空闲空间管理方法

* 位示图
  * 磁盘空间通常使用固定大小的块，可方便地用位示图管理，用若干字节构成一张位示图，其中每一字位对应一个物理块，字位的次序与块的相对次序一致，字位为‘1’表示相应块已占用，字位为‘0’表示该块空闲。
  * 微型机操作系统VM/SP、Windows和Macintosh等操作系统均使用这种技术管理文件存储空间。
  * 主要优点是：每个盘块仅需1个附加位，如盘块长1KB，位示图开销仅占0.012%；可把位示图全部或大部分保存在主存，再配合现代机器都具有的位操作指令，实现高速物理块分配和去配。
* 空闲区表
* 空闲块链

# 磁盘空闲空间管理方法

* 位示图
* 空闲区表
  * 该方法常用于连续文件，将空闲存储块的位置及其连续空闲的块数构成一张表。
  * 分配时，系统依次扫描空闲区表，寻找合适的空闲块并修改登记项；
  * 删除文件释放空闲区时，把空闲区位置及连续的空闲区长度填入空闲区表，出现邻接的空闲区时，还需执行合并操作并修改登记项。
  * 空闲区表的搜索算法有首次适应、邻近适应、最佳适应和最坏适应算法等，参见pp.239。
* 空闲块链

分配算法？

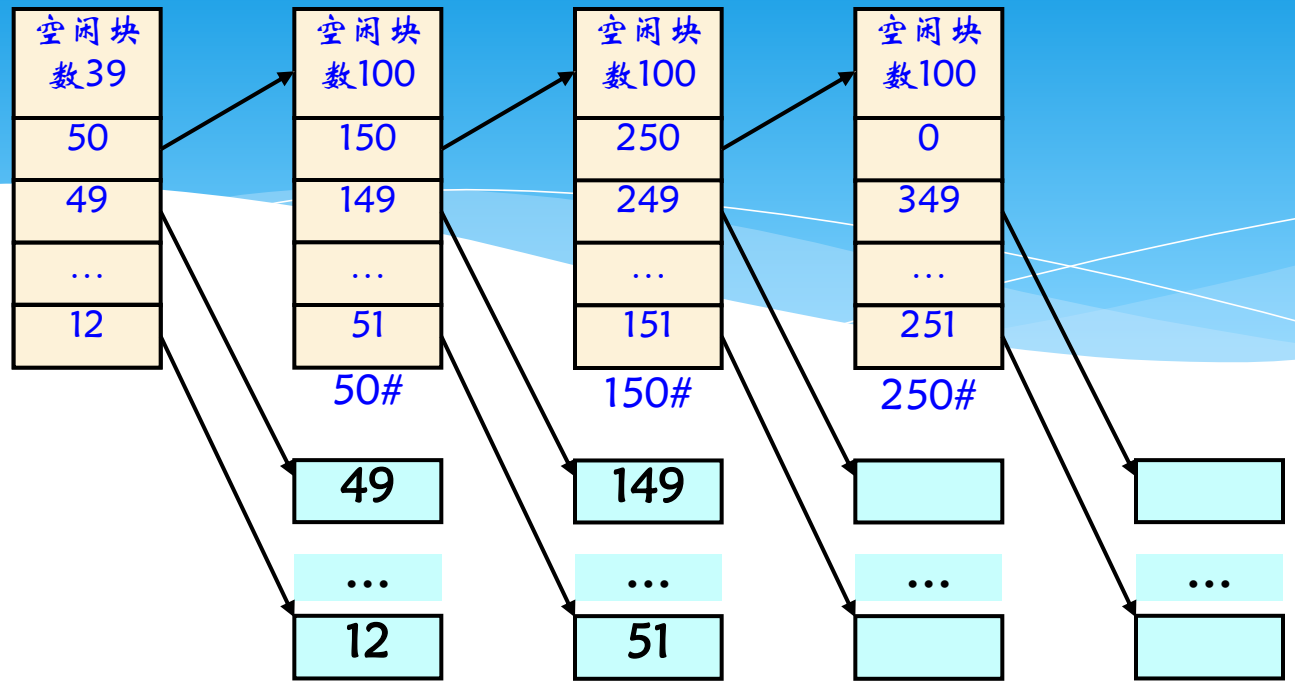
# 磁盘空闲存空间管理方法

* 位示图
* 空闲区表
* 空闲块链
  * 把所有空闲块连接在一起，系统保持指针指向第一个空闲块，每一空闲块中包含指向下一空闲块的指针。
  * 申请一块时，从链头取一块并修改系统指针；
  * 删除时释放占用块，使其成为空闲块并将它挂到空闲链上。
  * 这种方法效率低，每申请一块都要读出空闲块并取得指针，申请多块时要多次读盘，但便于文件动态增长和收缩。

# UNIX/Linux空闲块成组连接法(1)

* 存储空间分成512字节一块。假定文件卷启用时共有可用文件338块，编号从12至349。每100块划分一组，每组第一块登记下一组空闲块的盘物理块号和空闲总数。

# UNIX/Linux空闲块成组连接法(2)




(磁盘)专用块←→(主存)专用块

分配算法
IF 空闲块数=1 THEN
 IF第一个单元=0 THEN 等待
 ELSE 复制第一个单元对应块到专用块，并分配之
ELSE 分配第(空闲块数)个单元对应块，空闲块数减1

归还算法
IF 空闲块数<100
THEN 专用块的空闲块数加一，第(空闲块数)个单元置归还块号
ELSE 复制专用块到归还块，
 专用块的空闲块数置一，
 第一单元置归还块号

# 3.4主存映射文件(1)

* 首先，用于读写文件的操作在功能与格式上与读写主存的操作有很大不同，如果能消除这种差异就能简化编程工作；
* 其次，文件中的数据是一部分一部分在进程空间与磁盘空间之间传送，文件操作实现不但管理复杂且开销较大，能否找出一种方法既降低开销，又能通过直接读写主存来使用文件信息呢?
* 针对这一点，最早由MULTICS首创通过结合虚存管理和文件管理技术来提供一种新的文件使用方法，称主存映射文件，UNIX/Linux及Windows等现代操作系统都已实现这一功能。

# 3.4主存映射文件(2)

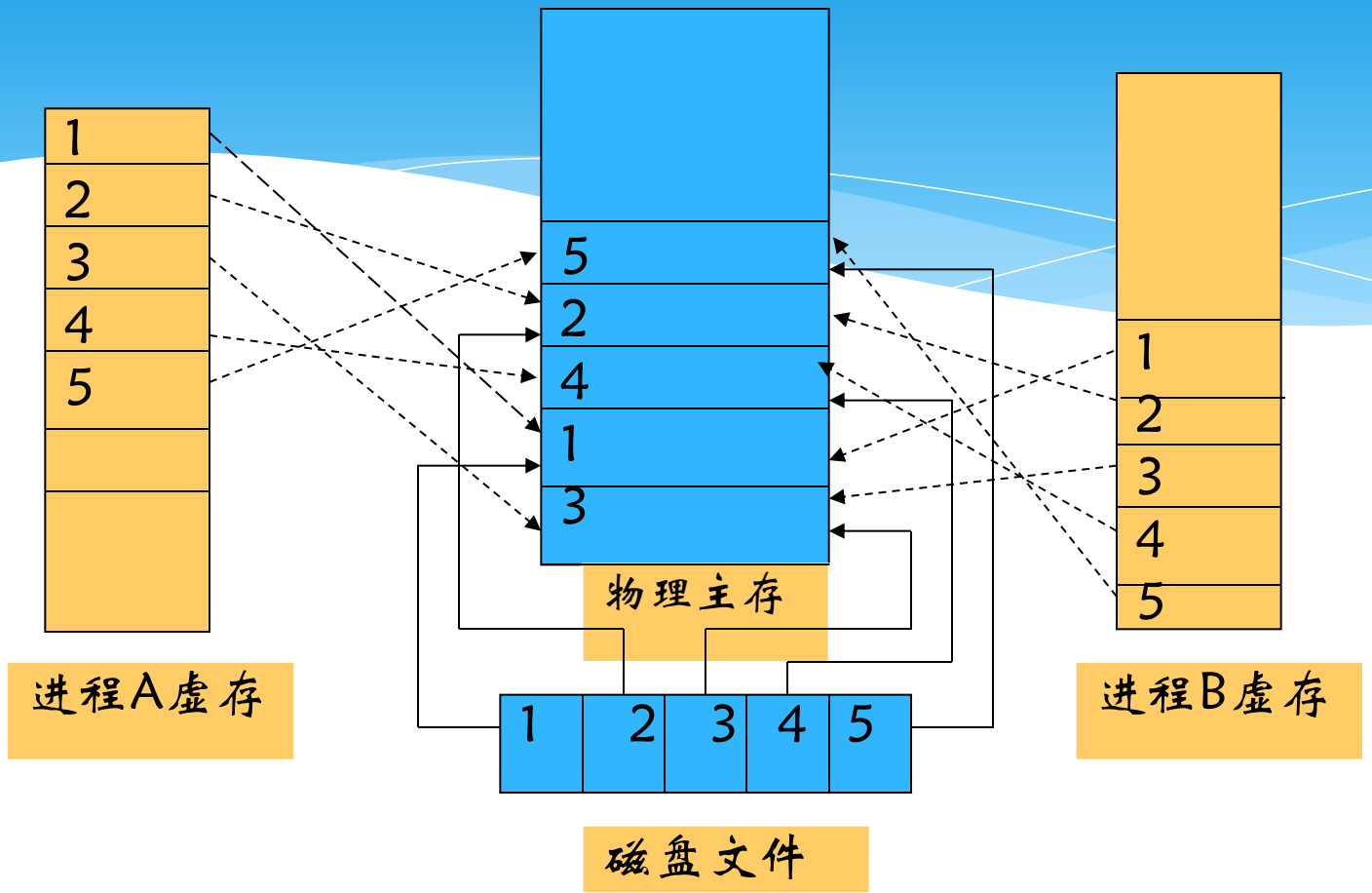
* 什么是主存映射文件
* 系统提供两个新的系统调用，

1)映射文件，有两个参数：一个文件名和一个虚拟地址，把一个文件映射到进程地址空间。

2)移去映射文件，让文件与进程地址空间断开，并把映射文件的数据写回磁盘文件。

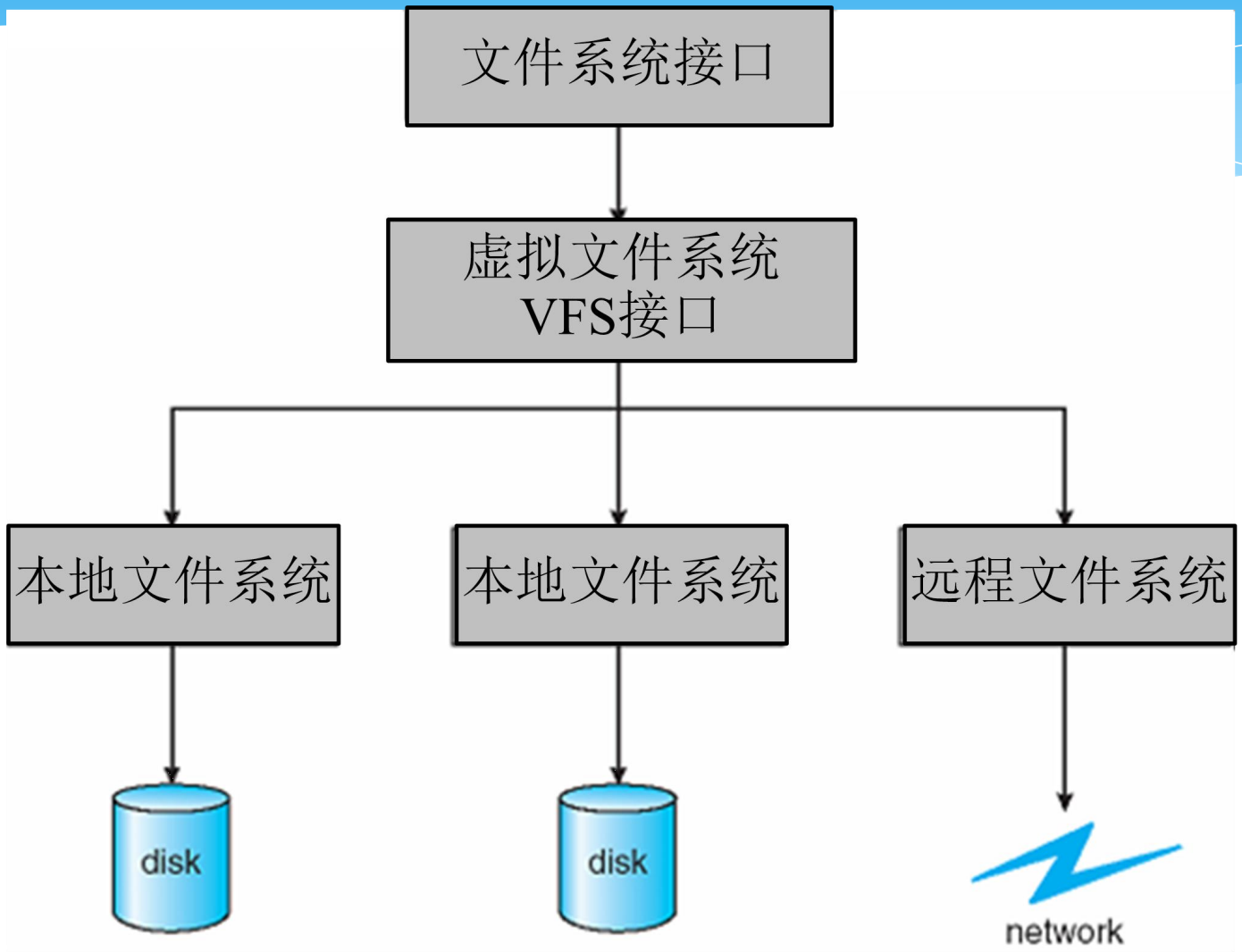
优点是：方便易用、节省空间、便于共享、灵活高效

# 主存映射文件(3)




优点：进程读写虚存内容相当于执行文件读写操作，在建立映射后，不再需要使用文件系统调用来读写数据，能大大降低开销；在主存中仅需一个页面副本，既节省空间，又不需要缓冲到主存的复制操作。

# 文件系统的系统视图


文件系统接口
虚拟文件系统
VFS接口
本地文件系统
本地文件系统
远程文件系统
disk
disk
network

# 虚拟文件系统(1)

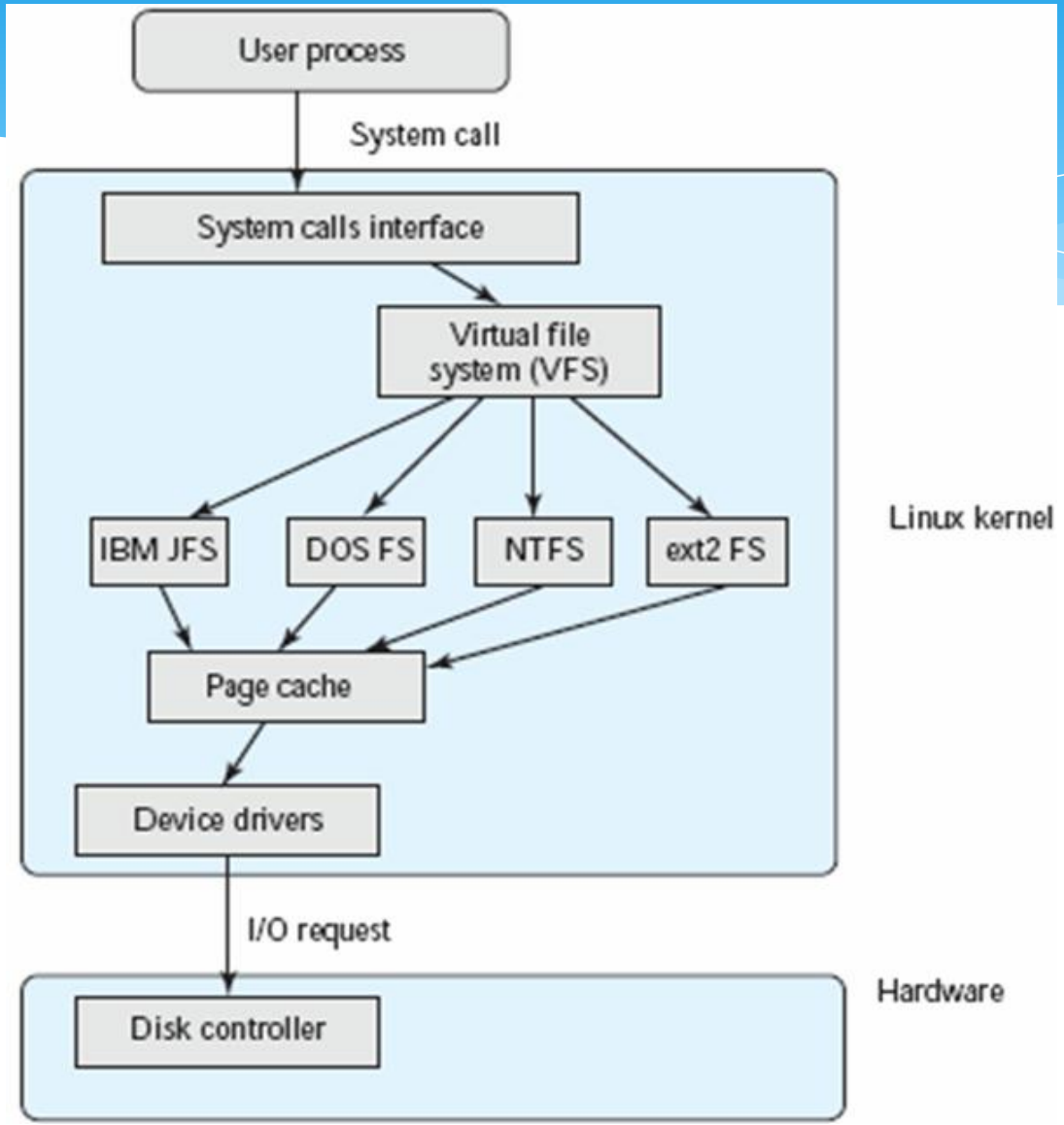
* 第一个虚拟文件系统在1986年由Sun公司开发成功，并在SunOS中使用。
* 虚拟文件系统也称虚拟文件系统开关VFS (Virtual Filesystem Switch)，
* 它是内核的一个子系统，提供一个通用文件系统模型，该模型概括所能见到的文件系统常用功能和行为，处理一切和底层设备驱动相关的细节，为应用程序提供标准的接口(文件系统API)。

# 3.5虚拟文件系统(1)

## 虚拟文件系统要实现以下目标

* 同时支持多种文件系统；
* 多个文件系统应与传统的单一文件系统没有区别，在用户面前表现为一致的接口；
* 提供通过网络共享文件的支持，访问远程结点上的文件系统应与访问本地结点的文件系统一致；
* 可以开发出新的文件系统，以模块方式加入到操作系统中。

# Linux虚拟文件系统



# 虚拟文件系统(2)

## 虚拟文件系统设计思想：

1 应用层：

VFS模型源于UNIX文件系统，使得用户可直接使用标准UNIX文件系统调用来操作文件，无需考虑具体文件系统特性和物理存储介质，通过VFS访问文件系统，才使得不同文件系统之间的协作性和通用性成为可能。

2 虚拟层：

3 实现层：

# 虚拟文件系统(2)

## 虚拟文件系统设计思想：

1 应用层：

2 虚拟层：

对所有具体文件系统的共同特性进行抽象的基础上，形成一个与具体文件系统实现无关的虚拟层，并在此层次上定义与用户的一致性接口；

3 实现层：

# 虚拟文件系统(2)

## 虚拟文件系统设计思想：

1 应用层：

2 虚拟层：

3 实现层：

该层使用类似开关表技术进行具体文件系统转接，实现各种具体文件系统的细节，每一个是自包含的，包含文件系统实现的各种设施，如超级块、节点区、数据区以及各种数据结构和文件类的操作函数。

# 虚拟文件系统(3)

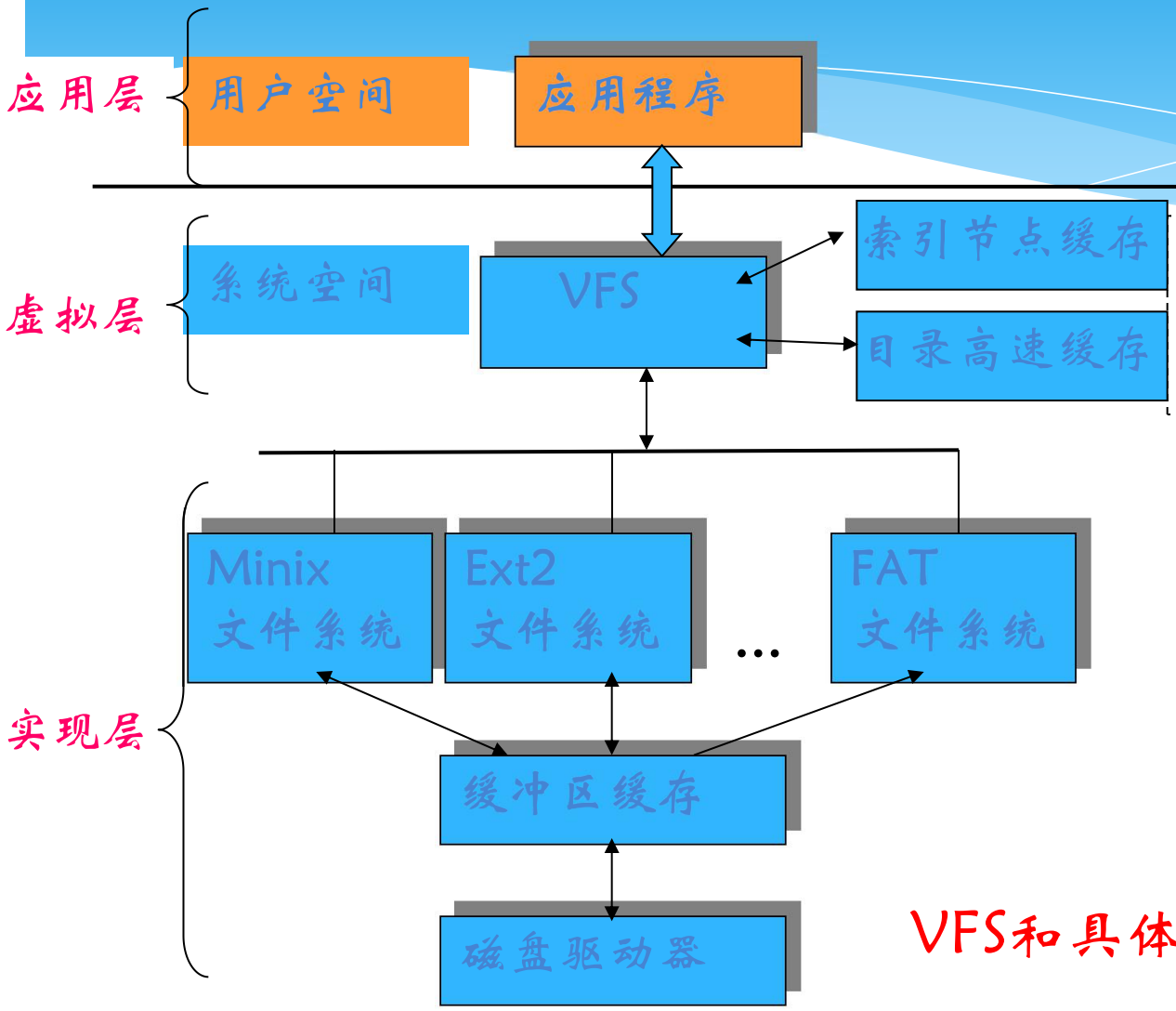
VFS实质上是一种存在于主存中的，支持多种类型具体文件系统的运行环境，功能有：

* 记录安装的文件系统类型，；
* 建立设备与文件系统的联系；
* 实现面向文件的通用操作；
* 涉及特定文件系统的操作时映射到具体文件系统中去。

# Linux的文件管理

* Linux虚拟文件系统
* 文件系统注册与注销，安装与卸载
* 文件系统缓存机制
* Ext2文件系统

# Linux虚拟文件系统




标准文件类系统调用，open()、read()、write()、close()等

系统调用对应的内核函数，sys_open()、sys_read()、sys_write()、sys_close()

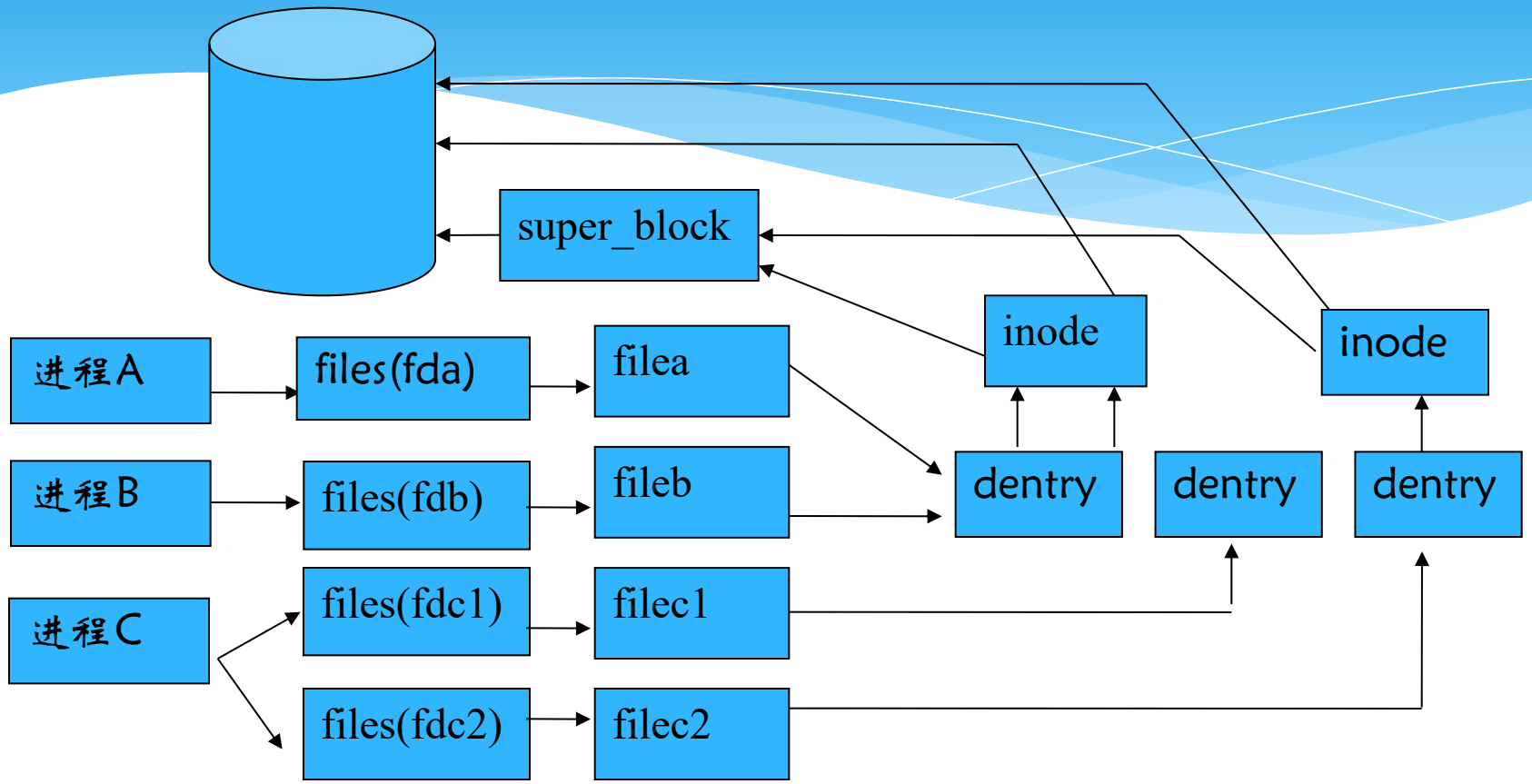
具体文件系统对应的文件操作函数，open()、read()、write()、close()等

VFS和具体文件系统的关系

# VFS的组成(主要数据结构)

* 超级块对象-代表一个文件系统。
* 索引节点对象-代表一个文件。
* 目录项对象-代表路径中的一个组成部分。
* 文件对象-代表由进程已打开的一个文件。

# VFS各种对象之间的关系




VFS各种对象之间的关系

# VFS的组成(主要数据结构)

* 超级块(super block)对象-代表一个文件系统。
  * 存放已安装的文件系统信息，如果是基于磁盘的文件系统，该对象便对应于存放在磁盘上的文件系统控制块，每个文件系统都对应一个超级块对象。
  * 如Ext2超级块，并被存放在磁盘特定扇区上，当内核对一个具体文件系统进行初始化和注册时，调用alloc_super( )函数为其分配一个VFS超级块，并从磁盘读取具体文件系统超级块中的信息填充进来，即VFS超级块在具体文件系统安装时才建立，并在它们卸载时被自动删除，可见VFS超级块仅存于主存中。
* 索引节点对象
* 目录项对象
* 文件对象

# 超级块

```
struct super_block {
 struct list_head s_list;          //把所有超级块双向链接起来
 kdev_t s_dev;                     // 文件系统所在设备标识符
 unsigned long  s_blocksize;       //以字节为单位的盘块大小
 unsigned char  s_blocksize_bits;// 以2幂次表示的块大小，如块为4KB，值为12
 unsigned long long s_maxbytes;  //文件大小上限
 unsigned char  s_dirt;            //修改(脏)标志
 …
 struct file_system_type  *s_type; // 指向注册表file_system_type结构的指针
 struct super_operations  *s_op;   // 指向超级块操作函数集的指针，
 struct dentry  *s_root;           //安装目录的目录项对象
 struct rw_semaphore s_umount;  //卸载信号量
 struct semaphore s_lock;          //超级块信号量
 int s_count;                      //超级块引用计数
 struct list_head s_dirty;         //脏节点inode链表
 struct list_head s_io;            //回写链表
 char s_id[32];                    //文本名字
 …
 union {  //一个联合体，成员是各个具体文件系统的fsname_sb_info数据结构
     struct minix_sb_info minix_sb;
…
  } u;
};
```

# VFS的组成(主要数据结构)

* 超级块(super block)对象
* 索引节点对象-代表一个文件
  * 存放通用的文件信息，如果是基于磁盘的文件系统，该对象对应于存放在磁盘上的文件FCB，即每个文件的inode对象，而每个inode都有一个inode索引节点号，唯一地标识某个文件系统中的指定文件。
  * 对于UNIX类文件系统来说，这些信息从磁盘inode直接读入VFS的inode对象中。可把具体文件系统存放在磁盘上的inode称为静态节点，它的内容被读入主存VFS的inode才能工作，后者也称为动态节点
* 目录项对象
* 文件对象

# 索引节点

```
struct inode {
  struct list_head i_hash;  //散列值相同的inode链表
  struct list_head i_list;      //指向inode链表的指针
  struct list_head i_dentry; //同属一个inode的dentry链表
  unsigned long i_ino;      //inode号
  kdevt idev;                  // 所在设备的设备号
  umode_t i_mode;          //文件类型及存取权限
  nlink_t i_nlink;           //连接到该inode的硬连接数
  uid_t i_uid;                 //文件拥有者的用户ID
  gid_t i_gid;                 //用户所在组的ID
  loff_t i_size;               //字节为单位文件大小
...
  struct semaphore;                //inode信号量
  struct inode_operations  *i_op; //指向inode操作函数的指针
  struct super_block  *i_sb;      //指向该文件系统超级块的指针
  atomic_t i_count;               //当前使用该inode的引用计数，0表示空闲
  atomic_t i_writecount;          //写者计数
  struct file_operations  *i_fop; //指向文件操作函数的指针
  time_t i_atime; i_mtime; i_ctime; //最近访问时间/修改时间/创建时间
  struct pipe_inode_info *i_pipe; //管道信息
  unsigned long  i_state;             //inode状态标志
  unsigned int i_flags;               //文件系统标志
...
  union
  {    //联合体成员指向具体文件系统的inode结构
      struct minix_inode_info minix_i;
...
}u;
};
```

# VFS的组成(主要数据结构)

* 超级块(super block)对象
* 索引节点对象
* 目录项对象-代表路径中的一个组成部分。
  * 存放目录项与对应文件进行链接的各种信息，VFS把最近最常使用的dentry对象放在目录项高速缓存中，加快文件路径名搜索过程，以提高系统性能。
* 文件对象

# 目录项

```
struct dentry {
  atomic_t d_count;              //目录项引用计数
  unsigned long d_vfs_flags;   //目录项缓存标志
  unsigned int d_flags;          //目录项状态标志
  struct inode * d_inode;        //dentry所属的inode
  struct dentry * d_parent;      //父目录的目录项对象
  struct list_head d_hash;       //目录项形成的哈希表
  struct list_head d_child;      //父目录的子目录项形成双向链表
  struct list_head d_subdirs;     //该目录项的子目录的双向链表
  struct list_head d_alias;      //inode别名的链表
  int d_mounted;                 //是安装点目录项吗?
  struct qstr d_name;            //目录项名字，用于快速查找
  unsigned long d_time;          //重新生效时间
  struct dentry_operations *d_op;  //操作目录项的函数
  struct super_block *d_sb;      //指向文件的超级块
  struct hlist_node d_hash;      //哈希表
  struct hlist_head *d_bucket;    //哈希表头
  unsigned char d_iname［DNAME_INLINE_LEN］;//文件名前15个字符
  …
};
```

# VFS的组成(主要数据结构)

* 超级块(super block)对象
* 索引节点对象
* 目录项对象
* 文件对象-代表由进程已打开的一个文件。
  * 存放已打开文件与进程的交互信息，这些信息仅当进程访问文件期间才存于主存中；
  * 文件对象在执行系统调用open()时创建，执行系统调用close()时撤销。

# VFS的组成(主要数据结构)

* 文件对象
  * 每个文件都用一个32位数字来表示下一个读写的字节位置，通常称它为文件位置或偏移量(offset)，每当打开一个文件时，偏移量被置0，读写操作便从这里开始，允许通过系统调用lseek对文件位置作随机定位。
  * Linux建立文件对象(file)来保存打开文件的文件位置，file结构除保存文件当前位置外，还把指向该文件inode的目录项指针也放在其中，并形成一个双向链表，称系统打开文件表。
  * 操作系统之所以不直接使用dentry结构是因为多个进程能够打开同一个文件，因为每一个file结构实际上对应了一个进程的一次打开过程。file结构中记录了文件访问模式，读写指针等信息。

# 系统打开文件表

```
struct file {
struct list_head f_list;        //所有打开文件形成的链表
struct dentry *f_dentry;        //指向相关目录项的指针
struct vfsmount *f_vfsmnt;      //指向VFS安装点的指针
struct file_operation *f_op;    //指向文件操作函数的指针
unsigned long f_reada;          //预读标志
unsigned long f_ramax;          //预读的最多页数
unsigned long f_raend;          //上次预读后的指针
unsigned long f_ralen;          //预读的字节数
unsigned long f_rawin;          //预读的页数
mode_t f_mode;                  //文件访问模式
loff_t f_pos;                   //文件当前偏移量
unsigned short f_count;         //使用该文件的进程数
unsigned int f_uid;             //使用者的用户标识
unsigned int f_gid;             //使用者的用户组标识
…
 };
```

# VFS的组成(主要数据结构)

* 文件对象
  * 文件描述符fd用来描述打开的文件，每个进程用一个files_struct结构来记录文件描述符的使用情况，这个结构称为用户打开文件表。
  * 指向该结构的指针被保存在进程的task_struct结构的成员files中。

# 用户打开文件表

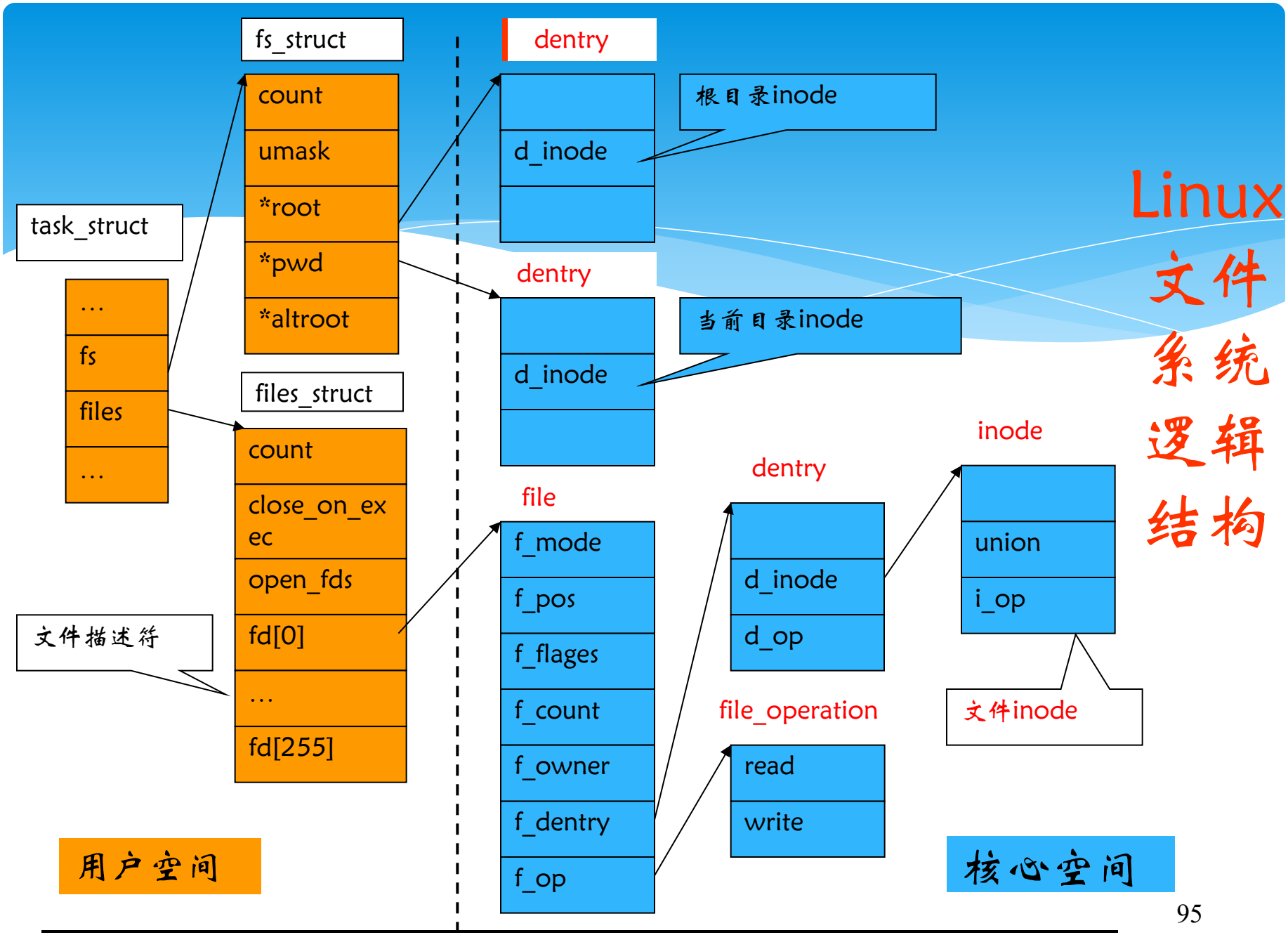
```
struct files_struct{
* atomic_t count;            //共享该表的进程数
* rwlock_t file_lock;        //保护该结构体的锁
* int max_fds;               //进程当前具有的最大文件数
* int max_fdset;             //当前文件描述符的最大数
* int next_fd;               //已分配的最大文件描述符加1
* struct file  **fd;         //指向文件对象(系统打开文件表项)的指针数组
* fd_set *close_on_exec;  //指向执行exec( )时需关闭的文件描述符
* fd_set *open_fds;         //指向打开文件的描述符的指针
* fd_set close_on_exec_init;  //执行exec( )时需关闭的文件描述符初值集
* fd_set open_fds_init;      //文件描述符初值集
* struct file *fd_array[32]; //指向文件对象的初始化指针数组
*  };
```

# 进程工作的文件系统信息(根目录和当前工作目录)—fs_struct结构

```
struct fs_struct {
  atomic_t count;          //共享fs_struct结构的进程数
  rwlock_t lock;           //保护此结构体的锁
  int umask;               //默认的文件访问权限掩码
  struct dentry *root;     //根目录的目录项对象
struct dentry *pwd,;       //当前工作目录的目录项对象
struct dentry *altroot;    //可替换的根目录的目录项对象
struct vfsmount *rootmnt;  //根目录的安装点对象
struct vfsmount *pwdmnt;   //当前工作目录的安装点对象
struct vfsmount *altrootmnt; //可替换的根目录的安装点对象
 };
```

# 与进程相关的文件系统数据结构

* 在进程的task_struct结构中有与文件系统相关的成员：
* struct task_struct {
*  …
*  struct fs_struct *fs;  //文件系统信息
*  struct files_struct *files;  //打开文件信息
*  …
* };
* 下面将要讨论系统打开文件表file结构、用户打开文件表files_struct结构和进程工作的根目录和当前工作目录信息fs_struct结构。


Linux
文件
系统
逻辑
结构
fs_struct
count
umask
*root
*pwd
*altroot
task_struct
…
fs
files
…
files_struct
count
close_on_exec
open_fds
fd[0]
…
fd[255]
文件描述符
dentry
d_inode
根目录inode
dentry
d_inode
当前目录inode
file
f_mode
f_pos
f_flages
f_count
f_owner
f_dentry
f_op
dentry
d_inode
d_op
file_operation
read
write
inode
union
i_op
文件inode
用户空间
核心空间

# 文件系统注册与注销，安装与卸载

1 文件系统的注册与注销

* Linux支持多个物理磁盘，每个磁盘可划分为一个或多个磁盘分区，每个分区上可建立一个文件系统，一个安装好的Linux操作系统究竟支持几种不同类型的文件系统，是通过文件系统类型注册链表来描述的，VFS以链表形式管理已注册的具体文件系统。
* 向系统注册文件系统类型有两种途径，
  * 一是在编译操作系统内核时确定可支持哪些文件系统，在文件系统被引导时，在VFS中进行注册；
  * 二是文件系统当作可装载模块，通过insmod/rmmod命令在装入该文件系统模块时向VFS注册/注销。

# 文件系统注册与注销，安装与卸载

1 文件系统的注册与注销

```
struct file_system_type {
    const char *name;/*文件类型名*/
    struct super_block *(*read_super)(struct super_block*,void*,int);
    struct file_system_type *next;
...
};
```

# 2 文件系统的安装与卸载

1文件系统安装

文件系统类型名、所在物理设备名、安装点，再用mount命令安装。

2文件系统安装过程

寻找匹配的file_system_type、查找安装点VFS inode、分配一个VFS超级块、利用read_super()函数读入参数、申请一个vfsmount数据结构。

3文件系统卸载过程

是否可卸载、如果为“脏”把VFS超级块写回磁盘、删去vfsmount。

# 文件系统的缓存机制

1 VFS inode缓存

把当前使用的inode采用散列技术保存起来，从中快速找到所需inode。

2 VFS目录高速缓存

系统维护表达路径与inode对应关系的VFS目录缓存，其中存放被访问过的目录 。

3 页高速缓冲区

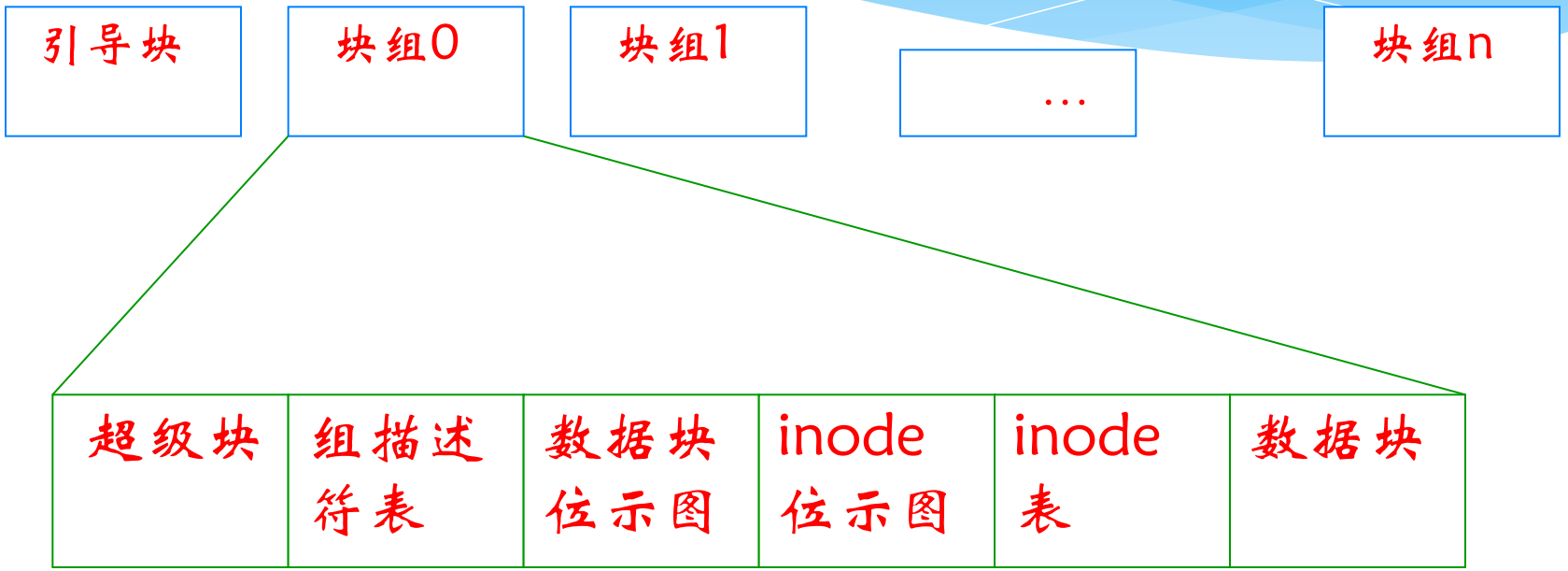
* Linux维护一组页缓冲区，它独立于任何类型的文件系统，被所有物理设备所共享，
* 优点：1)数据一经使用，就在页缓冲区中留下备份，再次使用时可直接找回，避免不必要的磁盘I/O；
* 2)“脏”页写回磁盘时，可适当进行排序，实现磁盘驱动调度优化。

# EXT2文件系统(1)

* EXT(92年)和EXT2(94年)是专为Linux设计的可扩展文件系统。
* EXT2把它所占用的磁盘逻辑分区划分为块组，每个块组依次包括超级块、组描述符表、块位示图、inode位示图、inode表以及数据块。
* 块位示图集中本组各数据块的使用情况；
* inode位示图记录inode表中inode的使用情况。
* inode表保存本组所有的inode，inode用于描述文件，一个inode对应一个文件和子目录，有一个唯一的inode号，并记录了文件在外存的位置、存取权限、修改时间、类型等信息。

# EXT2文件系统(2)

## 文件系统结构



# EXT2的超级块

* Ext2超级块用来描述目录和文件在磁盘上的静态分布，包括尺寸和结构，每个块组都有一个超级块，只有块1的超级块才被读入主存工作，直至卸载，其他块组的超级块仅作为恢复备份。
* 超级块主要包括：块组编号、块数量、块长度(1KB至4KB)、空闲块数量、inode数量、空闲inode数量、第一个inode号、第一个数据块位置、每个块组中的块数、每个块组的inode数，以及安装时间、最后一次写时间、安装信息、文件系统状态信息等内容。

# EXT2的组描述符

* 数据块位示图。表示数据块位示图占用的块号，此位示图反映块组中数据块的分配情况，在分配或释放数据块时需使用数据块位示图。
* inode位示图。表示inode位示图占用的块号，此位示图反映块组中inode的分配情况，在创建或删除文件时需使用inode位示图。
* inode表。块组中inode占用的数据块数，系统中的每个文件对应一个inode，每个inode都由一个数据结构来描述。
* 空闲块数、空闲inode数和已用数目。

# EXT2的inode

* inode用于描述文件，一个inode对应一个文件和子目录，有一个唯一的inode号，并记录了文件的类型及存取权限、用户和组标识、修改/访问/创建/删除时间、link数、文件长度和占用块数、在外存的位置、以及其他控制信息。

# Linux数据块分配策略(1)

EXT2采用两个策略减少文件碎片

* 原地先查找策略：为文件分配数据块时，尽量在文件原有数据块附近查找。先试探紧跟文件末尾的数据块，然后试探位于同一个块组相邻的64个数据块，接着在同一个块组中寻找其他空闲数据块；实在不得已才搜索其他块组，且首先考虑8个一簇的连续的块。

# Linux数据块分配策略(2)

* 预分配策略：引入预分配机制，就从预分配的数据块取一块来用，紧跟该块后的若干个数据块空闲的话，也被保留，保证尽可能多的数据块被集中成一簇。
* 数据结构中包含属性prealloc_block和prealloc_count，前者指向可预分配数据块链表中第一块的位置，后者表示可预分配数据块的总数。