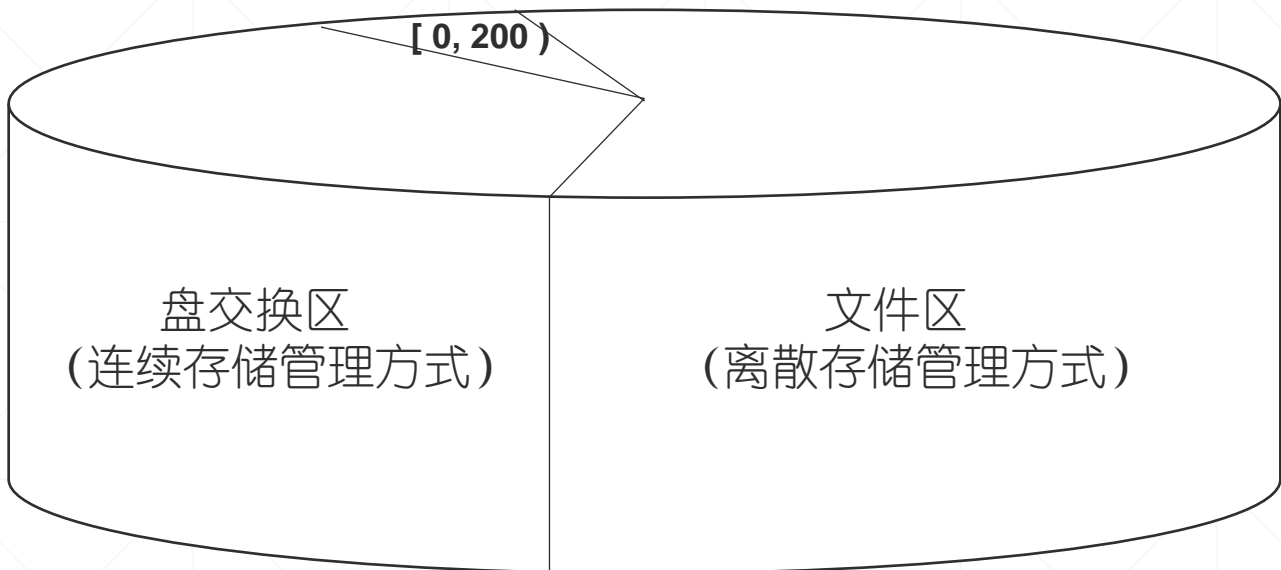


操作系统 第六章 文件管理

6.1 Unix V6++的文件系统静态结构

同济大学计算机系

一、磁盘 c.img

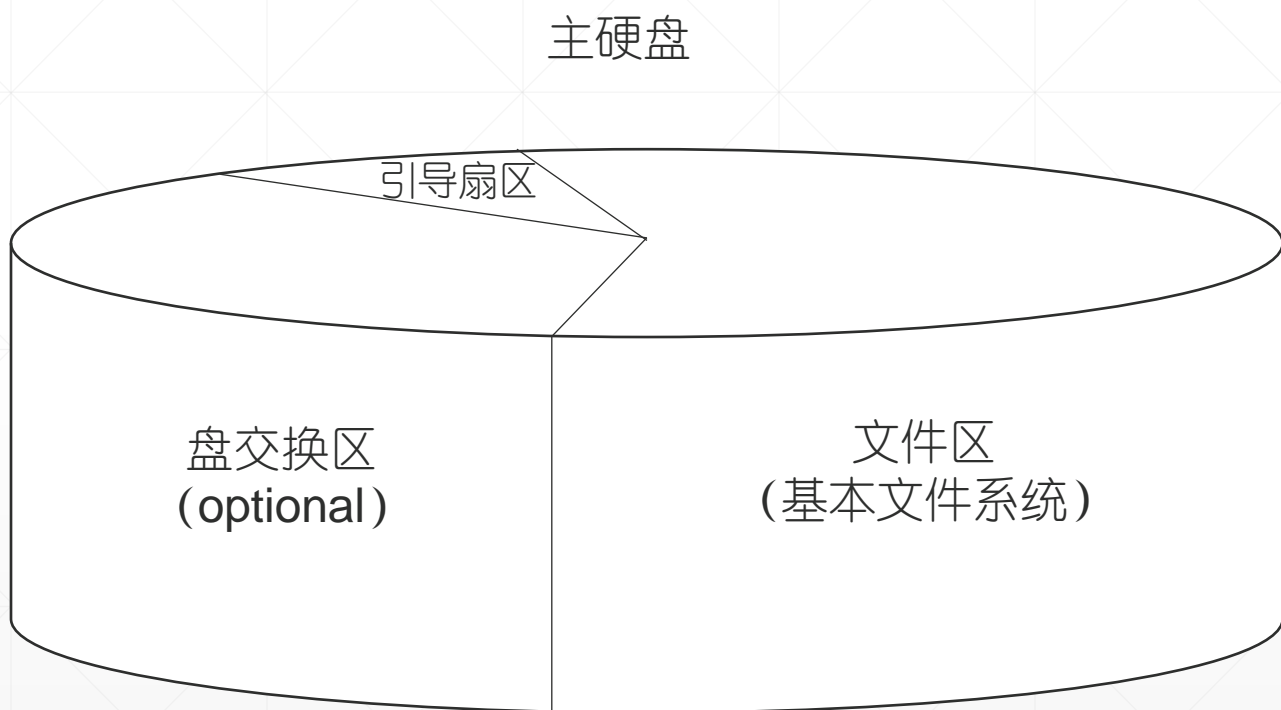


盘交换区放进程图像，连续存放。位置 [18000, 20000)。

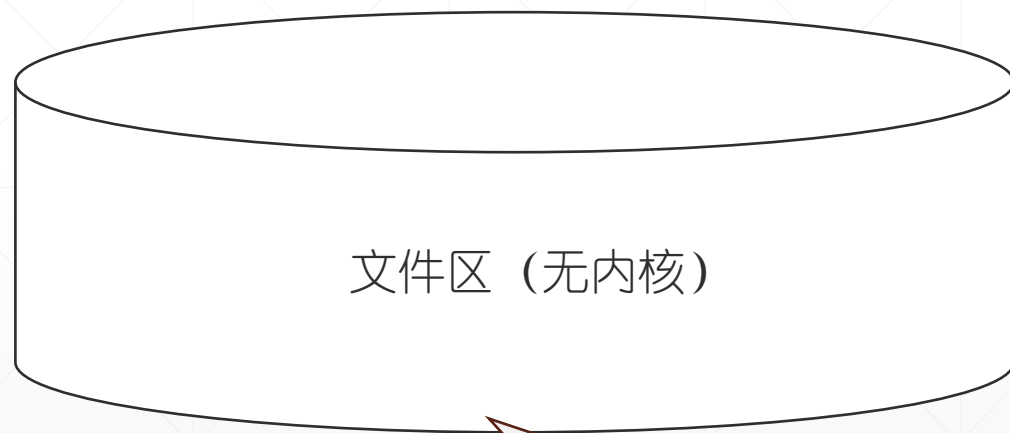
文件区存放文件。位置 [200, 18000)。文件分块，逻辑块的大小与磁盘物理块相等。文件离散存放，相邻逻辑块物理上可以不相邻，但连续存放一定可以提升文件访问性能。

0#扇区，引导扇区，boot程序，负责装载 kernel.exe
[1, 200)，内核 kernel.exe

一般而言，磁盘



普通的文件卷



以下，我们考虑Unix
V6++格式的普通文件卷



Unix V6+ + 格式的普通文件卷

0, 1

超级块	Inode区	文件区
-----	--------	-----

```
class SuperBlock
{
    int    s_isez;          /* Inode区尺寸，数据块数量 */
    int    s_fsize;        /* 磁盘数据块总数 */

    int    s_nfree;        /* 直接管理的空闲盘块数量 */
    int    s_free[100];    /* 直接管理的空闲盘块索引表 */
    int    s_flock;        /* 空闲盘块索引表的锁 */

    int    s_ninode;       /* 直接管理的空闲外存Inode数量 */
    int    s_inode[100];   /* 直接管理的空闲外存Inode索引表 */
    int    s_iloc;        /* 空闲Inode表的锁 */

    int    s_fmod;        /* 内存super block副本的修改标志，卸载时需要写回磁盘 */
    int    s_ronly;       /* 本文件系统只能读出 */
    int    s_time;        /* 最近一次更新时间 */
    int    padding[47];    /* 填充使SuperBlock块大小等于1024字节，占据2个扇区 */
};
```

磁盘空闲资源管理：
空闲盘块号栈
空闲Inode号栈



Unix V6+ 格式的普通文件卷





超级块 SuperBlock



```
class SuperBlock // 1024字节, 0#, 1#扇区
{
    int    s_isez;      /* Inode区尺寸, 数据块数量 */
    int    s_fsize;     /* 磁盘数据块总数 */

    int    s_nfree;     /* 直接管理的空闲盘块数量 */
    int    s_free[100]; /* 直接管理的空闲盘块索引表 */
    int    s_flock;     /* 空闲盘块索引表的锁 */

    int    s_ninode; /* 直接管理的空闲外存Inode数量 */
    int    s_inode[100]; /* 直接管理的空闲外存Inode索引表 */
    int    s_iloc;    /* 空闲Inode表的锁 */

    int    s_fmod;      /* 内存super block副本的修改标志, 卸载时需要写回磁盘 */
    int    s_ronly;     /* 本文件系统只能读出 */
    int    s_time;      /* 最近一次更新时间 */
    int    padding[47]; /* 填充使SuperBlock块大小等于1024字节, 占据2个扇区 */
};
```

磁盘空闲资源管理:

空闲盘块号栈
空闲Inode号栈

Inode 区



0	8						8n
1	9						8n+1
.....
7	15						8n+7

$n = s_isize - 1$
 分配单位, 64字节

```

class DiskNode // 64字节
{
public:
    unsigned int    d_mode;    /* 文件类型 和 访问控制位 */
    int            d_nlink;    /* 硬联结计数, 即该文件在目录树中不同路径名的数量 */
    short          d_uid;      /* 文件所有者的uid */
    short          d_gid;      /* 文件所有者的gid */

    int            d_size;      /* 文件长度, 字节为单位 */
    int            d_addr[10]; /* 地址映射表, 登记逻辑块和物理块之间的映射关系 */

    int            d_atime;     /* 最后访问时间 */
    int            d_mtime;     /* 最后修改时间 */
};
    
```



文件区



存放普通文件数据块，目录文件数据块，文件的索引块

- 普通文件数据块，文件数据
- 目录文件数据块，**16**个元素的目录项数组，每个元素对应一个等长目录项，**32**字节
- 文件索引块，**128**个元素的整数数组，每个元素对应一个逻辑块，其内容是一个分配给这个逻辑块的磁盘物理块号。**0**表示逻辑块是空的，尚无数据。

分配单位，数据块
(512字节)

普通磁盘文件

一个目录项 + 一个 Inode + 若干磁盘数据块

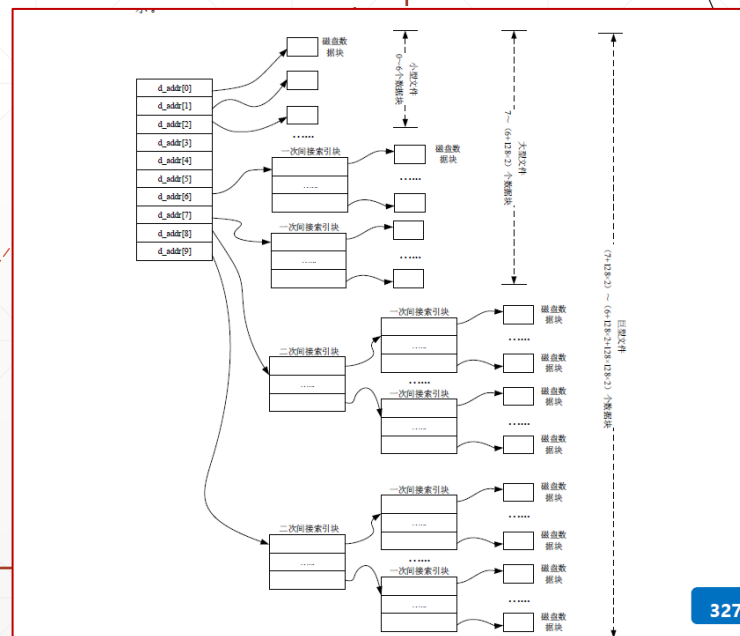
- 目录项在父目录文件里。
- 文件的 ID 是分配给这个文件的 DiskInode 的 ID。所以，foo 文件的 DiskInode 存放在 8号 inode 的位置：3#扇区，0~63字节。
- 分配给这个文件的磁盘数据块 组成一棵混合索引树，叶子节点装文件数据，中间节点装索引块。根是DiskInode中的地址映射表 d_addr。

例1：磁盘上的8#文件 foo



父目录的一个扇区

```
class DiskInode
{
public:
    unsigned int d_mode;
    int d_nlink;
    short d_uid;
    short d_gid;
    int d_size;
    int d_addr[10];
    int d_atime;
    int d_mtime;
};
```



foo文件的目录项

"foo\0"	8
---------	---

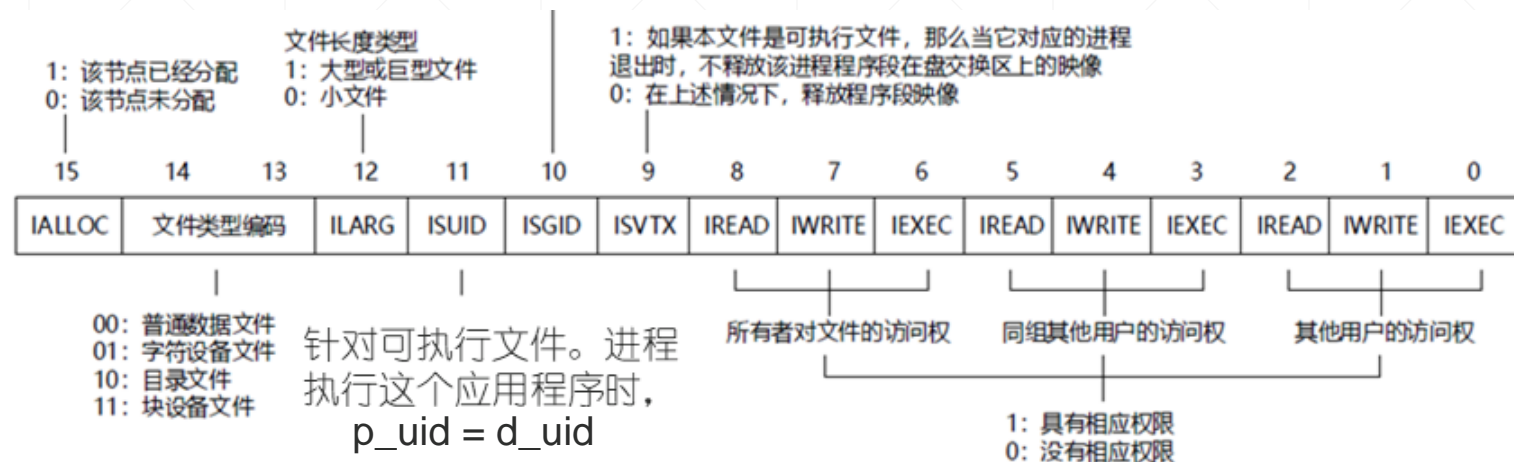
普通磁盘文件的DiskNode

`d_uid` = 执行 `create` 系统调用创建该文件的进程的 `p_uid`

```
class DiskNode
{
public:
    unsigned int d_mode;
    int d_nlink;
    short d_uid;
    short d_gid;
    int d_size;
    int d_addr[10];
    int d_atime;
    int d_mtime;
};
```

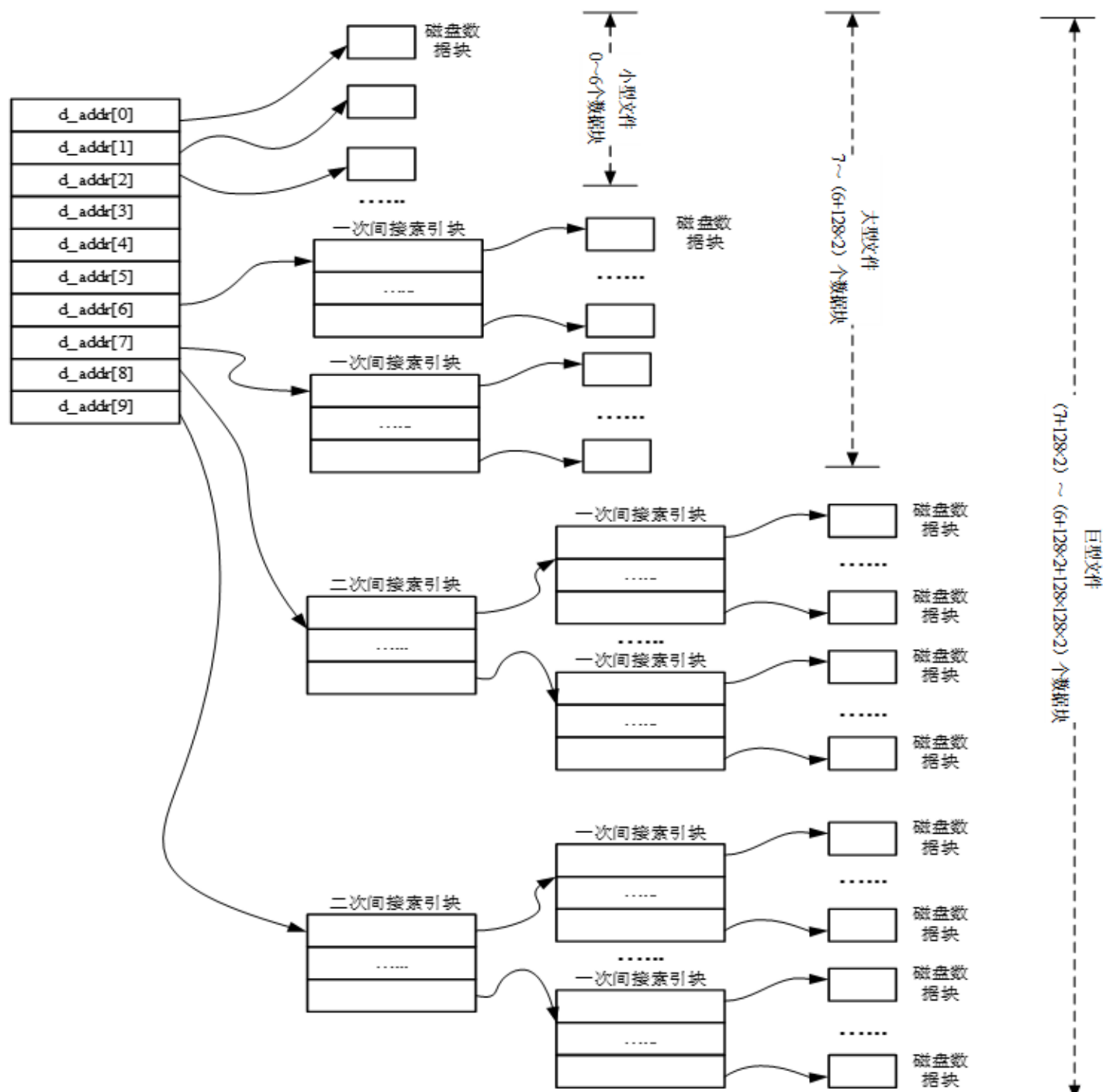
`creat(name, mode);`

RWXRWXRWX

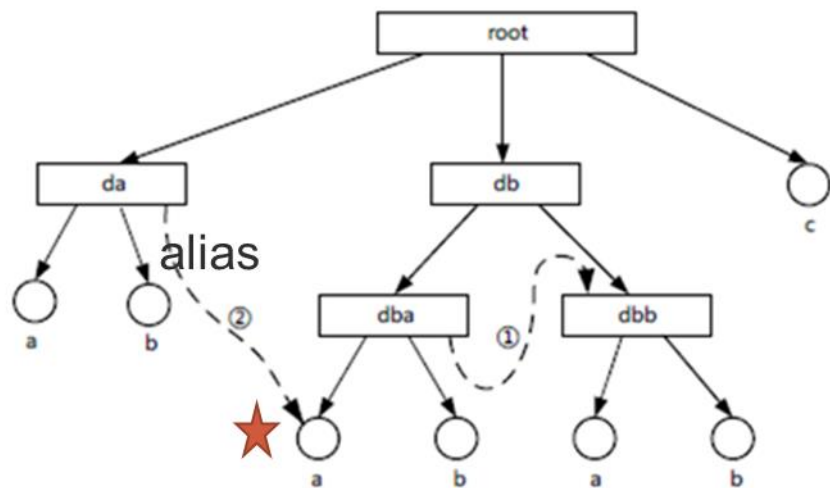


文件的混合索引树

```
class DiskInode
{
public:
    unsigned int d_mode;
    int d_nlink;
    short d_uid;
    short d_gid;
    int d_size;
    int d_addr[10];
    int d_atime;
    int d_mtime;
};
```



文件的硬链接数 d_nlink



100#文件

2个绝对路径名 (1) /da/alias (2) /db/dba/a

da目录文件中的目录项

"alias\0"	100
-----------	-----

dba目录文件中的目录项

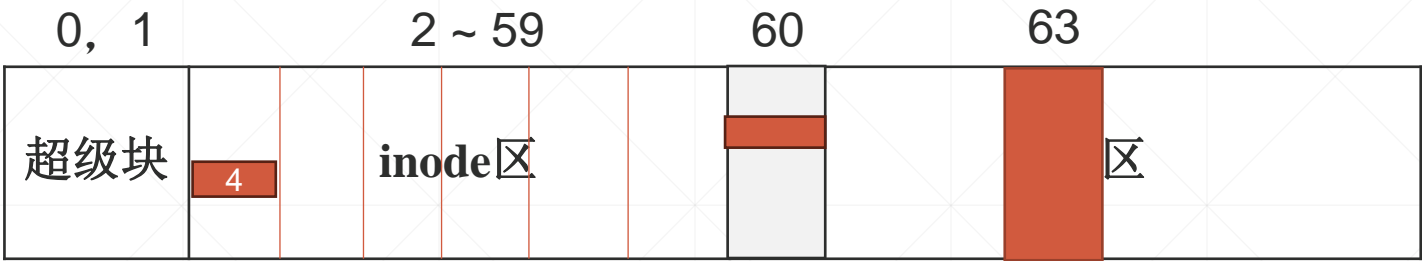
"a\0"	100
-------	-----

100#inode

d_link
(2)



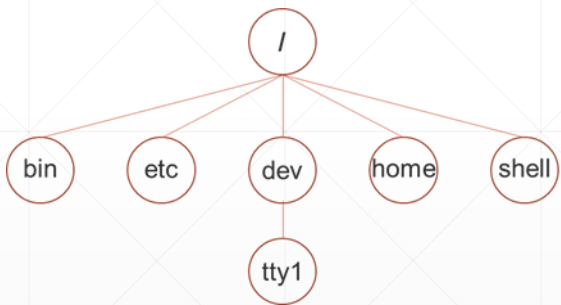
目录文件 /dev



```
class DirectoryEntry
{
    int m_ino; /* Inode号 */
    char m_name[DIRSIZ]; /* 路径名 */
};
```

class DiskInode

```
{
public:
    unsigned int d_mode; 10
    int d_nlink;
    short d_uid; 0
    short d_gid;
    int d_size;
    int d_addr[10]; d_addr[0] = 63
    int d_atime;
    int d_mtime;
};
```



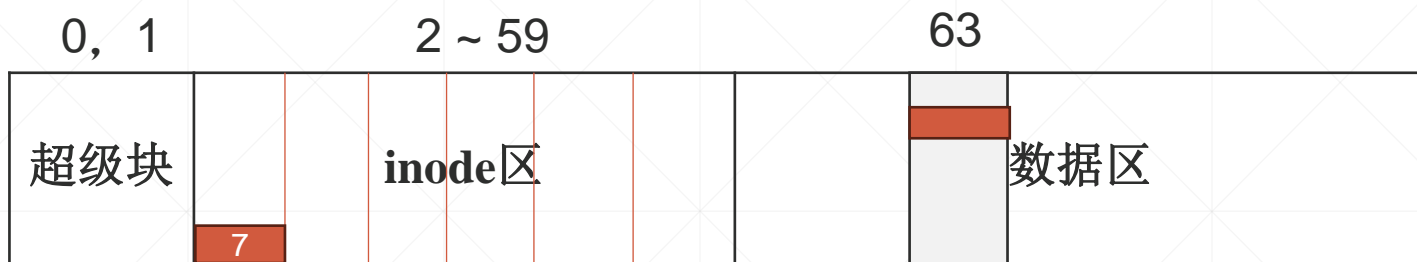
60#扇区

1	.\0
1	..\0
2	bin\0
3	etc\0
4	dev\0
5	home\0
6	shell\0
0	0
..... 0 0

63#扇区

4	.\0
1	..\0
7	tty1\0
0	0
..... 0 0

特殊文件：字符设备文件 /dev/tty1



```
class DiskInode
{
public:
```

```
    unsigned int d_mode; 01
    int d_nlink;
    short d_uid; 0
    short d_gid;
    int d_size;
    int d_addr[10];
    int d_atime; d_addr[0] = major, minor
    int d_mtime;
```

```
};
```

63#扇区(/dev目录文件)

4	.\0
1	..\0
7	tty1\0
0	0
..... 0 0

```
class DeviceManager
{
    BlockDevice *bdevsw[MAX_DEVICE_NUM];
    CharDevice *cdevsw[MAX_DEVICE_NUM];
};
```

```
class CharDevice
{
public:
    CharDevice();
    virtual ~CharDevice();
    /*
     * 定义为虚函数，由派生类进行override实现设备
     * 特定操作。正常情况下，基类中函数不应被调用到。
     */
    virtual void Open(short dev, int mode);
    virtual void Close(short dev, int mode);
    virtual void Read(short dev);
    virtual void Write(short dev);
    virtual void SgTTY(short dev, TTY* pTTY);

public:
    TTY* m_TTY; /* 指向字符设备TTY结构的指针 */
};
```

磁盘空闲资源



0	8						8n
1	9						8n+1
.....
7	15						8n+7

--	--	--	--	--	--	--	--

空闲 Inode 可以分配给新文件 或 新目录
分配: create, mkdir
释放: delete, rmdir

空闲盘块可以分配给 文件 或 目录
分配: write (文件), create、mkdir (目录)
释放: unlink (文件), unlink、rmdir (目录)

磁盘空闲资源管理



```
class SuperBlock
{
    .....
    int      s_nfree;          /* s_free中空闲盘块数量。s_free[-- s_nfree]是出栈操作 */
    int      s_free[100];      /* 空闲盘块号栈的首部 */
    int      s_flock;          /* 空闲盘块号栈的锁 */

    int      s_ninode;         /* 空闲 DiskInode 数量， s_inode[-- s_ninode]是出栈操作 */
    int      s_inode[100];     /* 空闲Inode栈，可以存放100个空闲 DiskInode号 */
    int      s_ilock;          /* 空闲Inode栈的锁 */
    .....
};
```


磁盘空闲资源管理

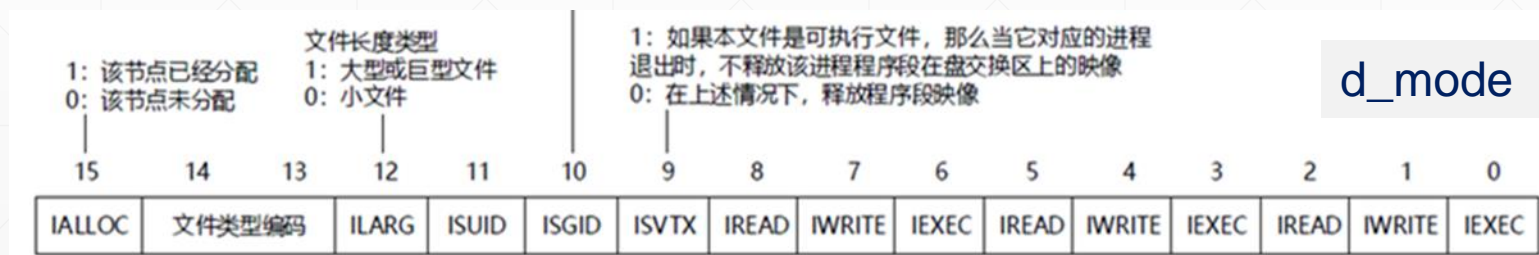


```
class SuperBlock
{
    .....
    int    s_nfree;
    int    s_free[100];
    int    s_flock;

    int    s_ninode;
    int    s_inode[100];
    int    s_iloc;
    .....
};
```

空闲盘块号栈很长，记录磁盘上所有的空闲盘块。
空闲Inode栈很小，只能存100个空闲 inode。

两者之间的差异，在于是否有办法从内容物判断一个单元是否空闲。



DiskInode空闲的判断条件：IALLOC是0 & 内存Inode池中没有这个Inode。

空闲 inode 管理

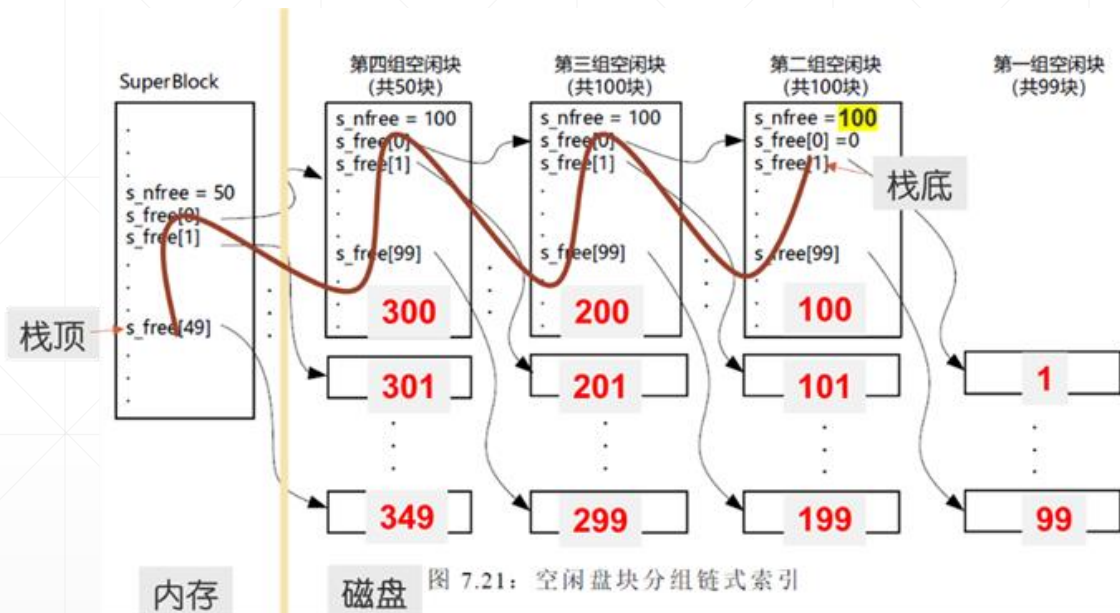
FileSystem::IAlloc(dev)

```
分配i节点: ialloc( dev), 取空闲i节点栈的栈顶节点  
again: if (s_ninode)  
    return(s_inode[--s_ninode]);  
else {  
    get next 100 free inodes from disk;  
    goto again; }
```

FileSystem::IFree(dev, num)

```
回收num#的i节点: ifree(dev,num)  
if (s_ninode != 100)  
    s_inode[s_ninode++] = num;
```

空闲盘块管理



利用Super Block和空闲盘块本身存放空闲盘块号栈
不消耗额外磁盘空间

FileSystem::Alloc(dev)

```
1、n = s_free[--s_nfree]; // pop
2、if (s_nfree==0)
    将n#扇区开头的404字节复制到 SuperBlock,
    覆盖 s_nfree和s_free;
3、return(n);
```

FileSystem::Free(dev, num)

```
if (s_nfree==100) //新组长空闲块num, 装入SuperBlock中的链接信息
{
    将 SuperBlock 中, s_nfree 和 s_free、s_nfree复制到num#扇区;
    s_nfree=0;
}
s_free[s_nfree++] = num; // pop
```

空闲资源管理 位示图 (bit map)

Unix 卷



- IB: inode bitmap (1个bit, 1个inode。 M 个bit)
- DB: data bitmap (1个bit, 1个物理块。 N个bit)

M是inode的总数
N是磁盘数据块的总数

操作系统 第六章 文件管理

6.2 Unix V6++ 文件系统的使用

同济大学计算机系



Part 1 普通文件的使用 1

文件描述符

`fd = open(文件名, mode);` // 用前打开。mode是文件打开方式
(1) 读 `FREAD` (2) 写 `FWRITE` (3) 读写 `FREAD | FWRITE`

文件访问

- `count = read(fd, &array, nbytes);`
- `count = write(fd, &array, nbytes);`
- `seek(fd, offset, ptrname);`

`close(fd);` // 用完关闭

2、文件的读写指针

一次文件访问会话

```
fd = open( 文件名, mode);
```

文件访问

- `count = read(fd, &array, nbytes);`
- `count = write(fd, &array, nbytes);`
- `seek(fd, offset, ptrname);`

```
close( fd ); // 用完关闭
```

进程用 `fd` 标识文件访问会话

每个文件访问会话有一根读写指针 `f_offset`

- `open` 返回时, `f_offset == 0`。
- `read / write` 系统调用从读写指针当前位置开始, 向后连续读/写 `nbytes` 个字符。读写操作完成之后 `f_offset += count`。

`seek (lseek)` , 调整读写指针的值

- `ptrname==0`, `f_offset = offset`
- `ptrname==1`, `f_offset += offset`
- `ptrname==2`, `f_offset = d_size + offset`



3、顺序读写 和 随机读写

(1) 顺序读写。程序不使用`seek`系统调用改变读写指针。`read/write` 在上次结束的位置往后读/写。

```
fd = open( file, mode);
```

文件访问

- `count = read(fd, &array, nbytes);`

.....

- `count = read(fd, &array, nbytes);`

.....

```
close( fd ); // 用完关闭
```

(2) 随机读写。程序使用`seek`系统调用设置下次`read/write` 系统调用的起始偏移量

```
fd = open( file, mode);
```

文件访问 (交织)

- `count = read(fd, &array, nbytes);`

- `count = write(fd, &array, nbytes);`

- `seek(fd, offset, ptrname);`

```
close( fd ); // 用完关闭
```


例：顺序读

```
fd = open( file, mode);
```

```
count = read( fd, &array, 1000); // [0,1000)
```

处理array中读到的文件数据

```
▪ count = read( fd, &array, 500); // [1000,1500)
```

处理array中读到的新数据

```
close( fd ); // 用完关闭
```

例：随机读

```
fd = open( file, mode);
```

```
count = read( fd, &array, 1000); // [0,1000)
```

处理array中读到的文件数据

```
seek( fd, 5000, 0);
```

```
count = read( fd, &array, 500); // [5000,5500)
```

处理array中读到的新数据

```
close( fd ); // 用完关闭
```

4、读写指针 是 会话 的属性

假设foobar.txt文件的内容是字符串“1234567890”。
请问，这个程序的输出是什么？

```
int main()
{
    int fd1, fd2;
    char c;

    fd1 = Open("foobar.txt", O_RDONLY, 0);
    fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd1, &c, 1);
    Read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

open, 打开文件、开启会话
create, 创建新文件、开启会话
close, 关闭会话

父进程创建子进程 (fork), 子进程继承父进程已开启的会话, 共享 f_offset (一份)

成熟的exec系统调用。有些会话系统调用完成后会保持, 有些需关闭。fd位图对此进行描述。

5、打开文件结构

```
class User
{
    .....
    OpenFiles u_files; // 进程的打开文件表
    .....
}
```

```
class OpenFiles
{
    .....
    File *ProcessOpenFileTable[NOFILES];
}
// 15, 进程可以同时访问的文件上限。文件可以相同
```

```
class File
{
    unsigned int f_flag; // 文件读写方式
    int f_count; // 引用计数, 是File结构的入边
    Inode* f_inode; // 指向分配给这个文件的内存Inode
    int f_offset; // 文件读写指针
};
```

(1) 读 FREAD (2) 写 FWRITE
(3) 读写 FREAD | FWRITE

```
class Inode
{
    short i_dev;
    int i_number;

    unsigned int i_flag; /* 状态的标志位, 定义见enum INodeFlag */
    int i_count; /* 引用计数 */

    int i_lastr; /* 最近读过的逻辑块号, 用于判断是否需要预读 */
    static int rablock; /* 预读块的物理块号 */

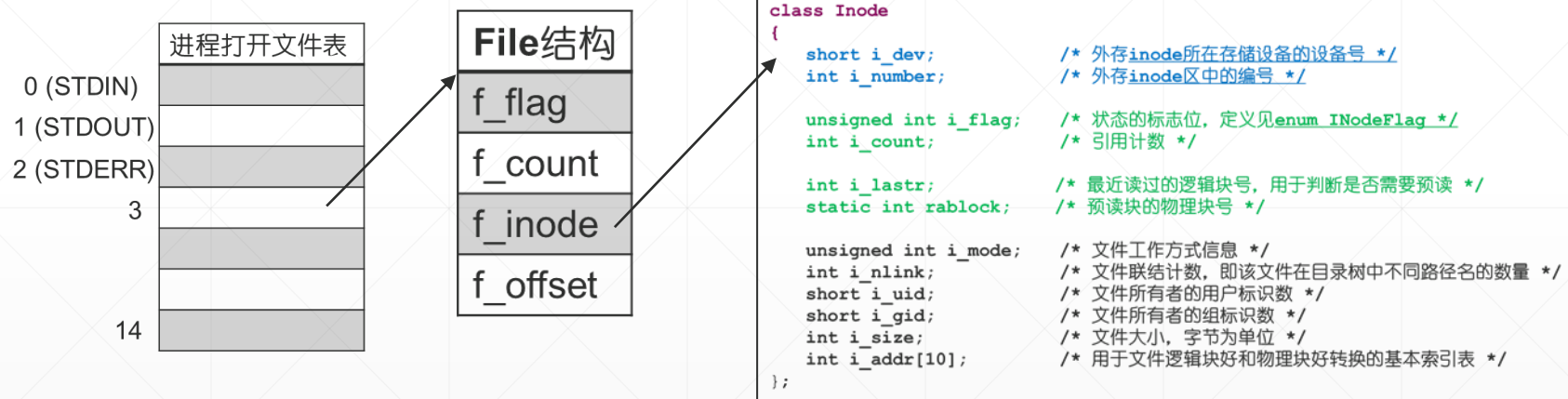
    unsigned int i_mode; /* 文件工作方式信息 */
    int i_nlink; /* 文件联结计数, 即该文件在目录树中不同路径名的数量 */
    short i_uid; /* 文件所有者的用户标识数 */
    short i_gid; /* 文件所有者的组标识数 */
    int i_size; /* 文件大小, 字节为单位 */
    int i_addr[10]; /* 用于文件逻辑块好和物理块好转换的基本索引表 */
};
```

DiskInode所在的磁盘和ID

Inode的使用状态

DiskInode的内存副本

进程的打开文件结构



文件描述符是进程打开文件表的下标

进程打开文件表 (进程的私有结构)

进程打开文件表	
0 (STDIN)	
1 (STDOUT)	
2 (STDERR)	
3	
14	

`open()`系统调用, `create()` 系统调用在打开文件表中为文件分配一个表项, 首次适应, 第1个非空表项。返回值`fd`是打开文件表的下标。

`fd`是文件描述符, 标识进程访问的第`fd`个文件。

- 0, STDIN。进程的标准输入
- 1, STDOUT。进程的标准输出
- 2, STDERR。进程的标准错误输出 (V6++没有)

- 这三个文件不是进程 (应用程序) 自己打开的。它们, 用户登录 (login) 时打开, fork时继承自父进程 (shell进程)
- V6没有登录过程, 系统初始化时`main0`打开这3个文件

文件打开后, 对该文件实施的`read`、`write`系统调用, 直接使用打开文件描述符`fd`。

例:

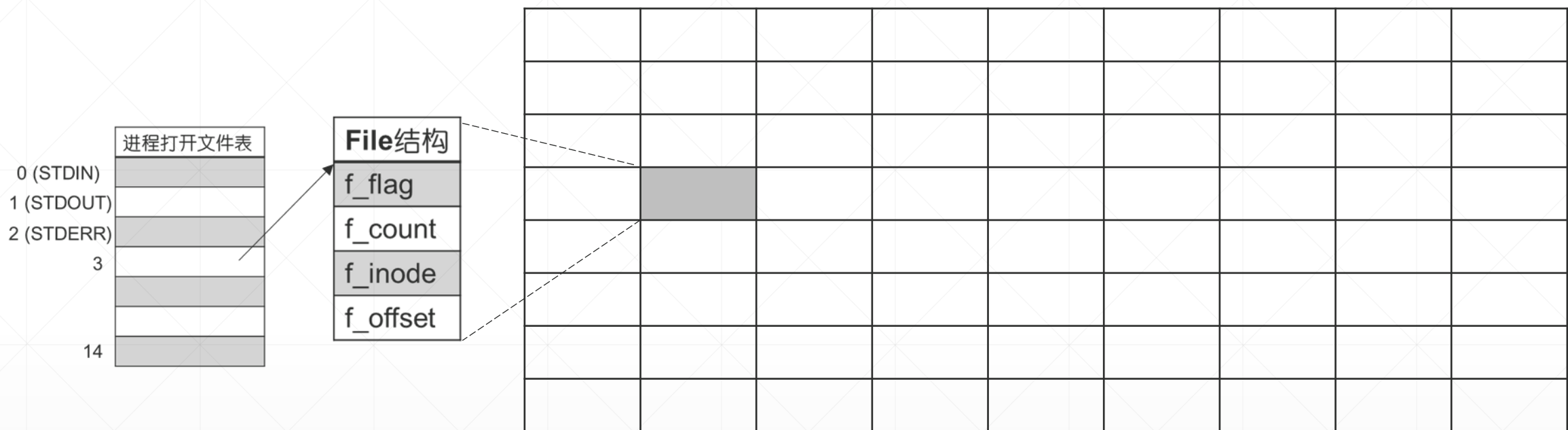
(1) fd1, fd2的值是几?

(2) 下面程序的输出是什么? 解释程序的输出。

```
int main()
{
    int fd1, fd2;

    fd1 = Open("foo.txt", O_RDONLY, 0);
    Close(fd1);
    fd2 = Open("baz.txt", O_RDONLY, 0);
    printf("fd2 = %d\n", fd2);
    exit(0);
}
```

File结构（系统范围内，供多个进程共享）

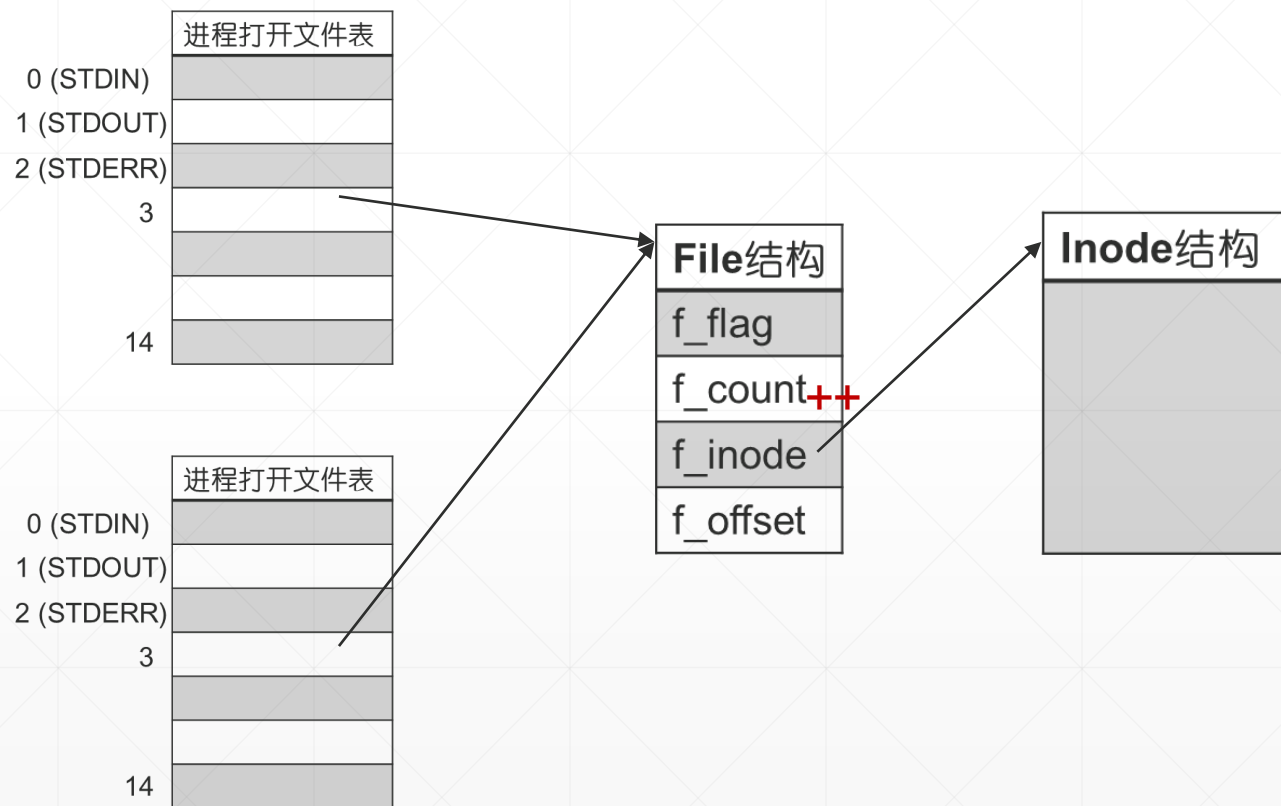


f_count==0, 空闲项

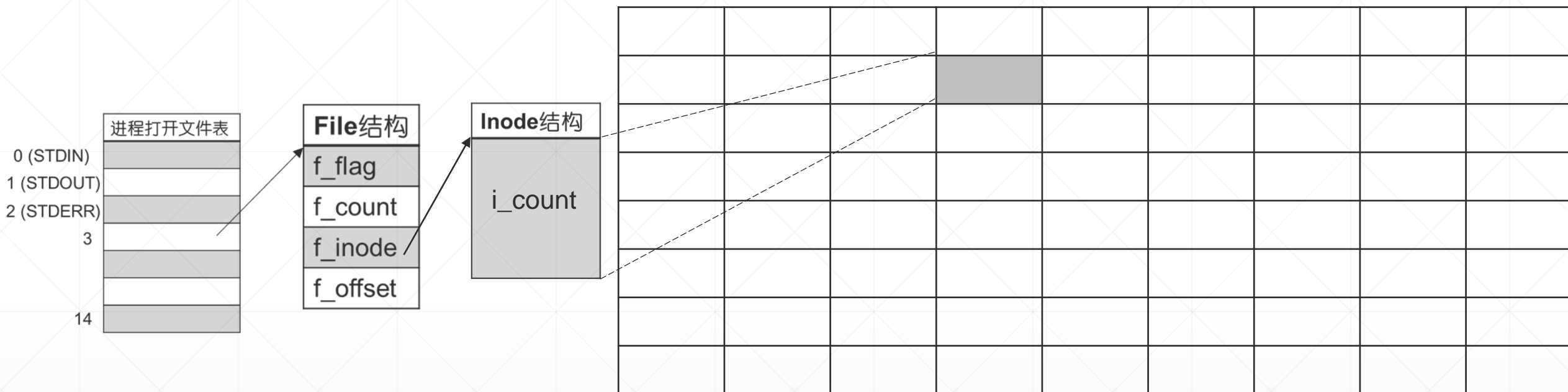
open、create会分配一个空闲File结构

```
class OpenFileTable
{
    File m_File[NFILE];    // 系统打开文件表, 100个元素
}
```


Fork, 父子进程引用同一个File结构



Inode结构（系统范围内，访问同一个文件的所有进程共享）



`i_count == 0`, 空闲项

`open`、`create`、`exec`

- 目标文件不命中，分配一个空闲Inode结构
- `i_count ++`

```
class InodeTable
```

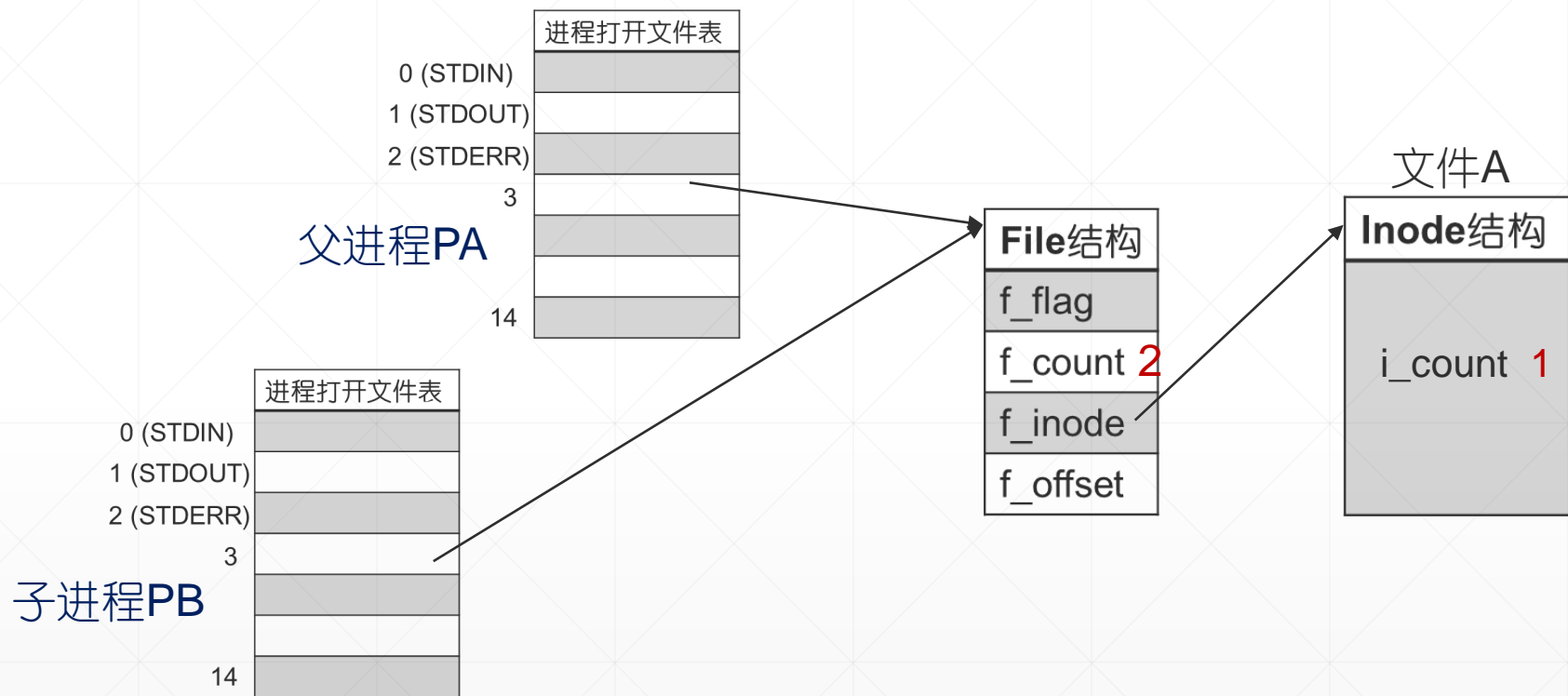
```
{
```

```
    Inode m_Inode[NINODE] // 内存Inode表，100个元素
```

```
}
```

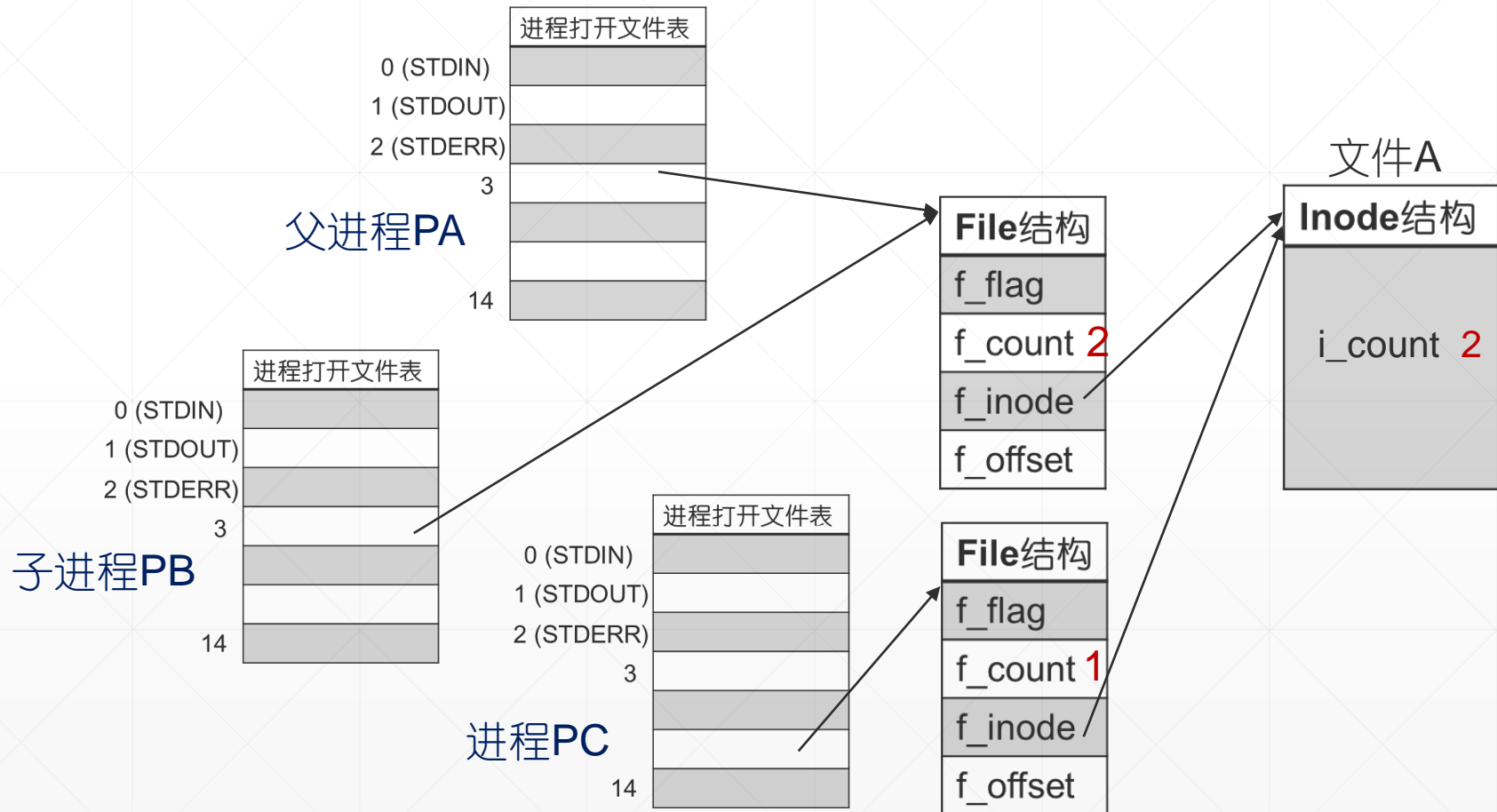
访问同一个文件的进程共享内存打开文件结构 1

父进程 open(文件 A) 之后创建子进程PB



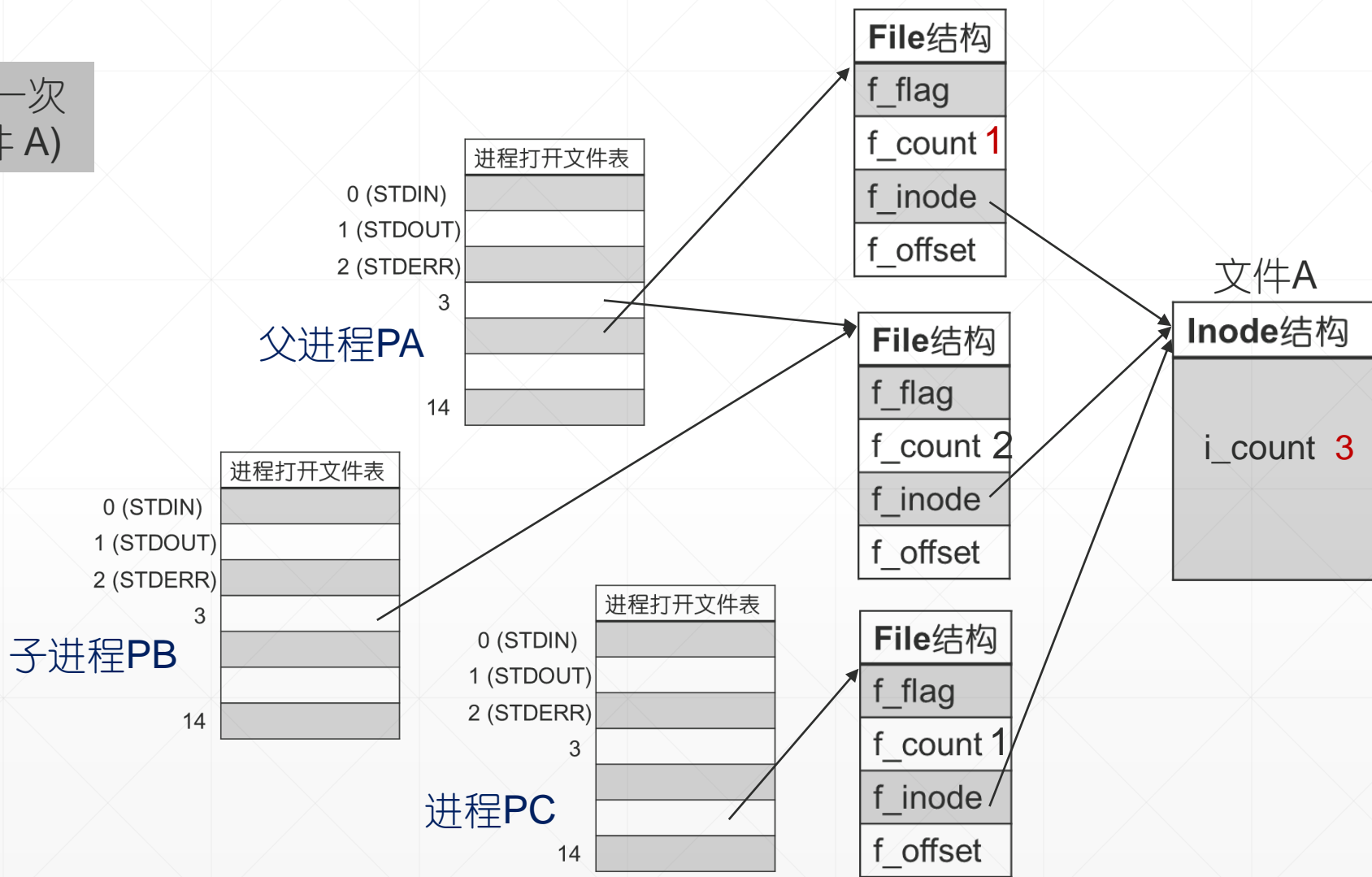
访问同一个文件的进程共享内存打开文件结构 2

进程PC执行系统调用 open(文件 A)



访问同一个文件的进程共享内存打开文件结构 3

进程PA，又执行了一次
系统调用 `open(文件 A)`



6、open系统调用

$fd = open(name, mode);$

- 目录搜索，确定文件 name 的 DiskInode 号。文件不存在，报错，返回。
- 为DiskInode分配内存Inode结构（用DiskInode 号搜索内存Inode表）
 - 命中，i_count++
 - 不命中，分配空闲Inode，读入磁盘Inode，初始化。
- 检查文件访问权限。Inode中，RWX不支持mode，报错，返回。
- 创建打开文件结构
 - 分配File结构，分配fd，建立fd→File→Inode勾连关系，设置引用计数器
- 返回文件描述符 fd

```
class Inode
{
    short i_dev;          /* 外存inode所在存储设备的设备号 */
    int i_number;         /* 外存inode区中的编号 */

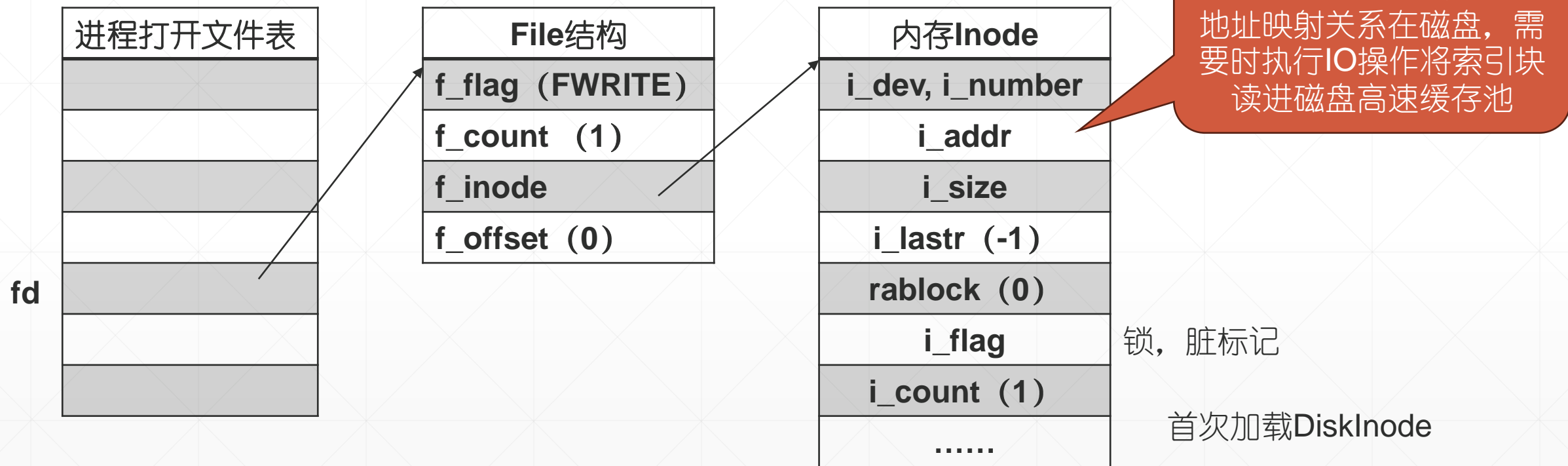
    unsigned int i_flag;  /* 状态的标志位，定义见enum INodeFlag */
    int i_count;          /* 引用计数 */

    int i_lastr;          /* 最近读过的逻辑块号，用于判断是否需要预读 */
    static int rablock;   /* 预读块的物理块号 */

    unsigned int i_mode;  /* 文件工作方式信息 */
    int i_nlink;          /* 文件联结计数，即该文件在目录树中不同路径名的数量 */
    short i_uid;          /* 文件所有者的用户标识数 */
    short i_gid;          /* 文件所有者的组标识数 */
    int i_size;           /* 文件大小，字节为单位 */
    int i_addr[10];       /* 用于文件逻辑块好和物理块好转换的基本索引表 */
};
```

Open 对打开文件结构的初始化

fd = open(name, mode);



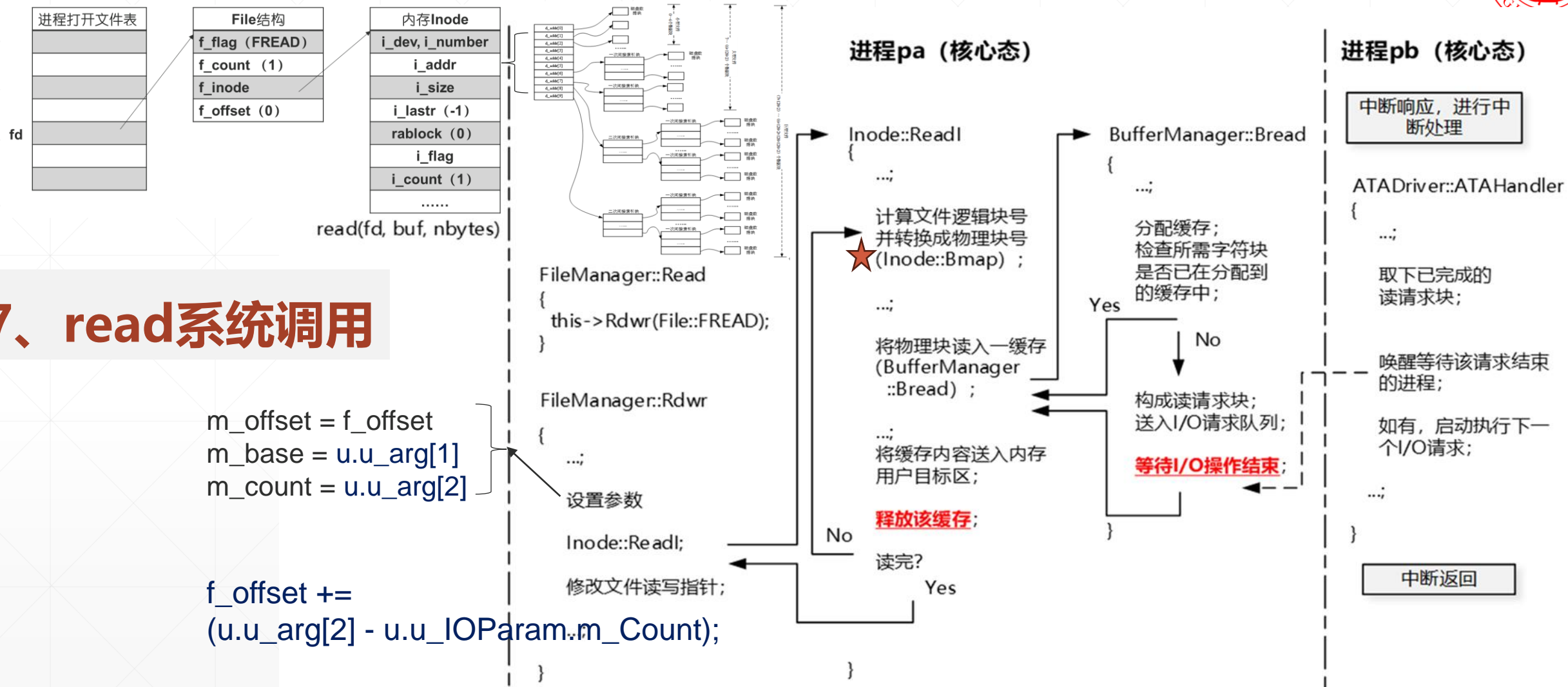


图 7.38: 系统调用 read 的基本执行过程



8、write系统调用

write(fd, buf, nbytes)

进程pa (核心态)

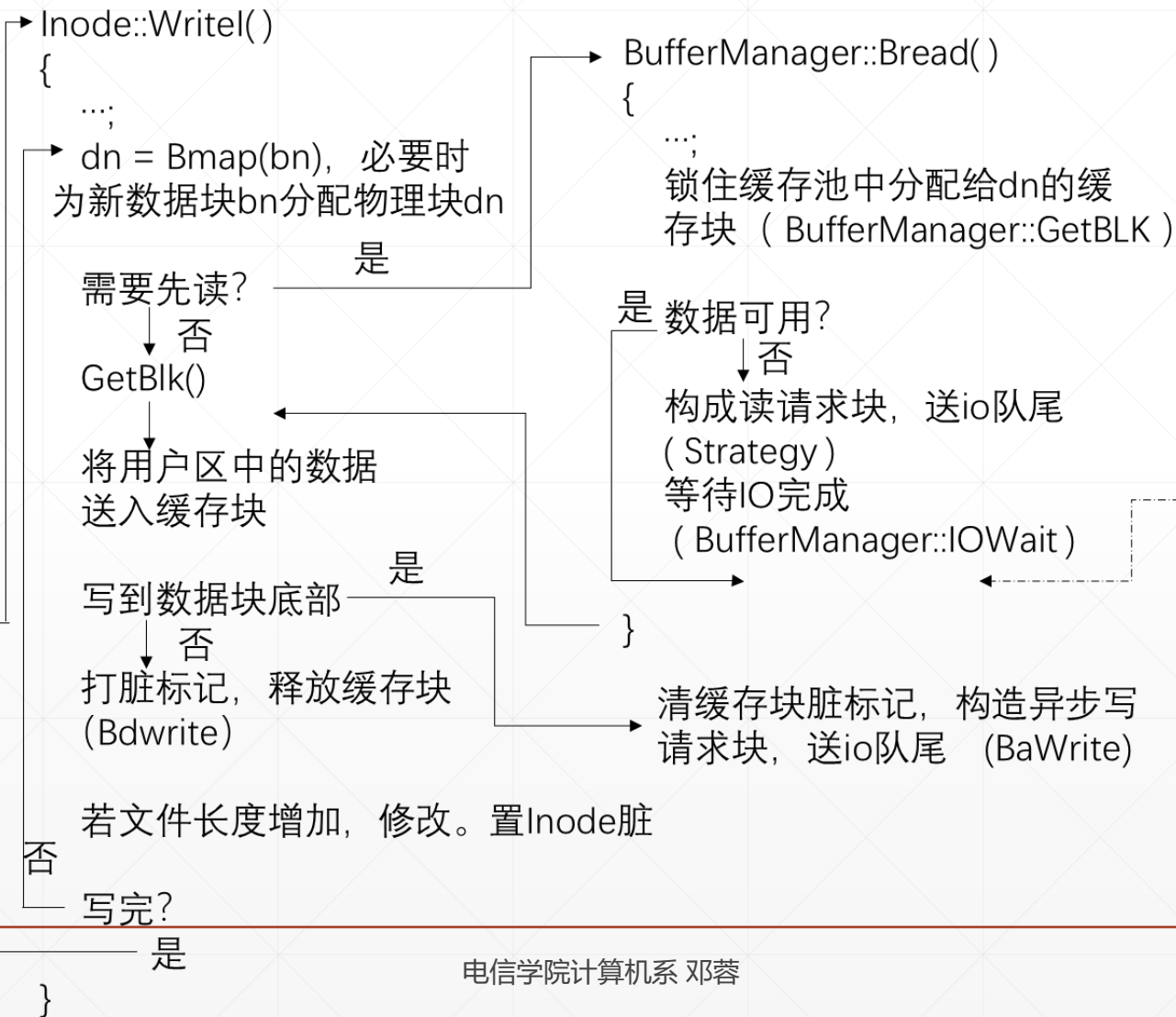
进程pb

(中断发生时的现运行进程)

系统调用传参

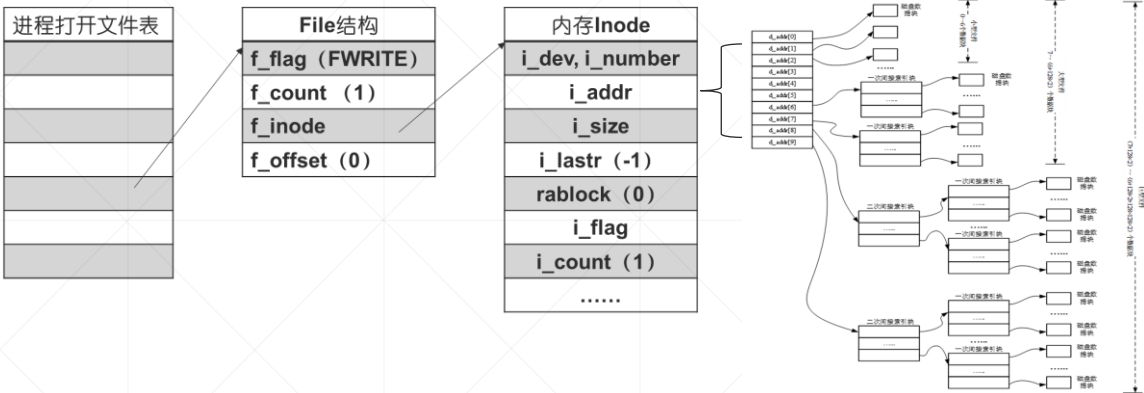
```
FileManager::Write()  
{  
    this->Rdwr(File::FWRITE);  
}
```

```
FileManager::Rdwr()  
{  
    ...;  
    设置IO参数;  
    Inode::Writel;  
    修改文件读写指针  
    ...;  
}
```



```
ATADriver::ATAHandler()  
{  
    ...;  
    摘除io队首的缓存块  
    置io完成标识B_DONE  
    同步块?  
    Y: 唤醒等待先读操作的进程  
    N: 释放缓存块  
    启动io队列中的下一个  
    io请求 (若队列不空)  
}
```

8、write系统调用



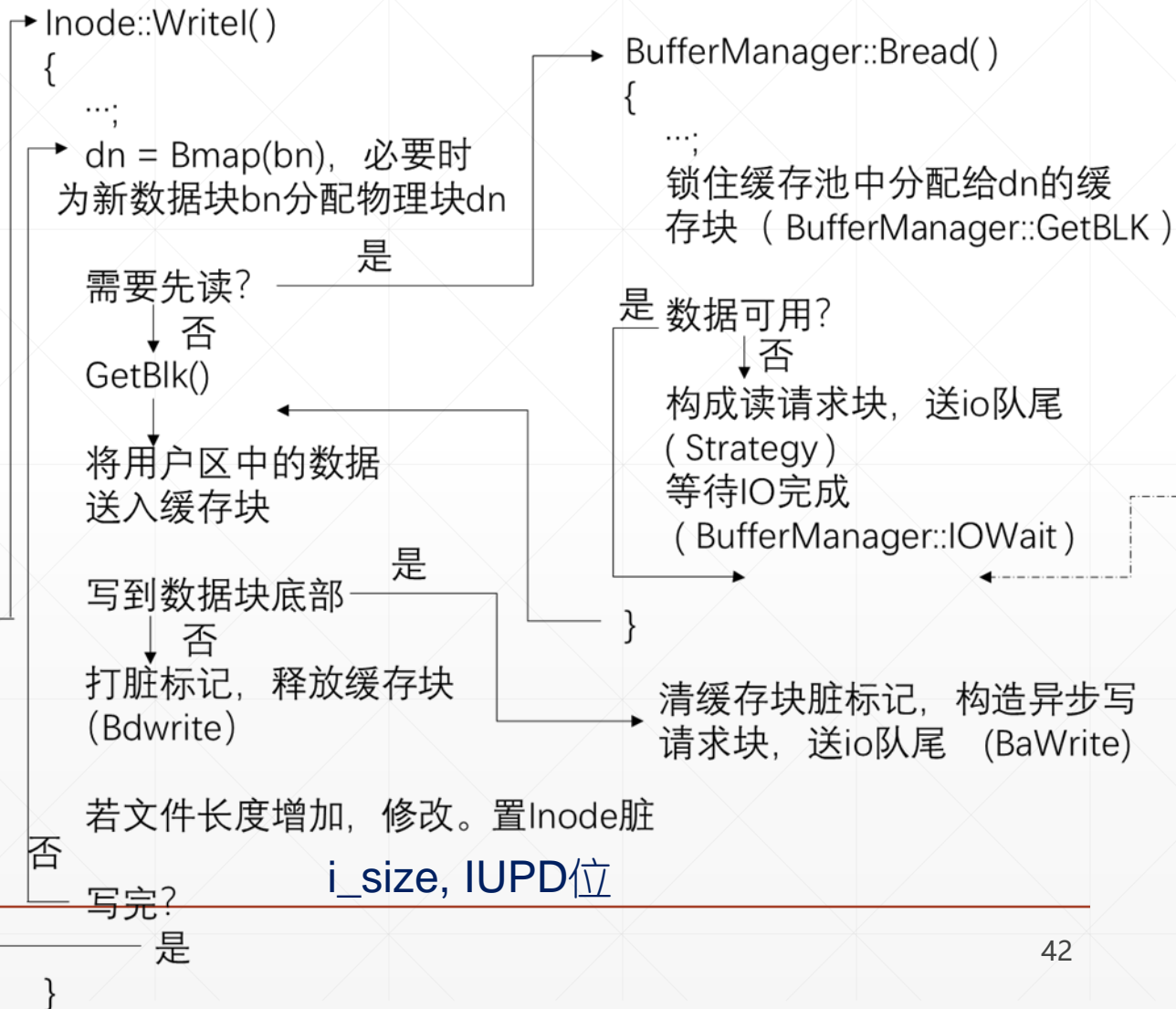
```
FileManager::Write()
{
    this->Rdwr(File::FWRITE);
}
```

```
m_offset = f_offset
m_base = u.u_arg[1]
m_count = u.u_arg[2]
```

```
FileManager::Rdwr()
{
    ...;
    设置IO参数;
    Inode::Writel;
    ...;
    修改文件读写指针
    ...;
}
```

```
f_offset +=
(u.u_arg[2] - u.u_IOParam.m_Count);
```

进程pa (核心态)



作业：一定要会写

分析代码7.11中所示实例的详细执行过程：

```
int fd = open("/usr/ast/Jerry",2); //以可读可写方式打开文件
char data[300];
seek(fd, 500, 0); //将文件读写指针定位到第500字节
int count = read (fd, data, 300); //从文件读300字节到data
count = write(fd, data, 300); //从data写300字节到文件
```

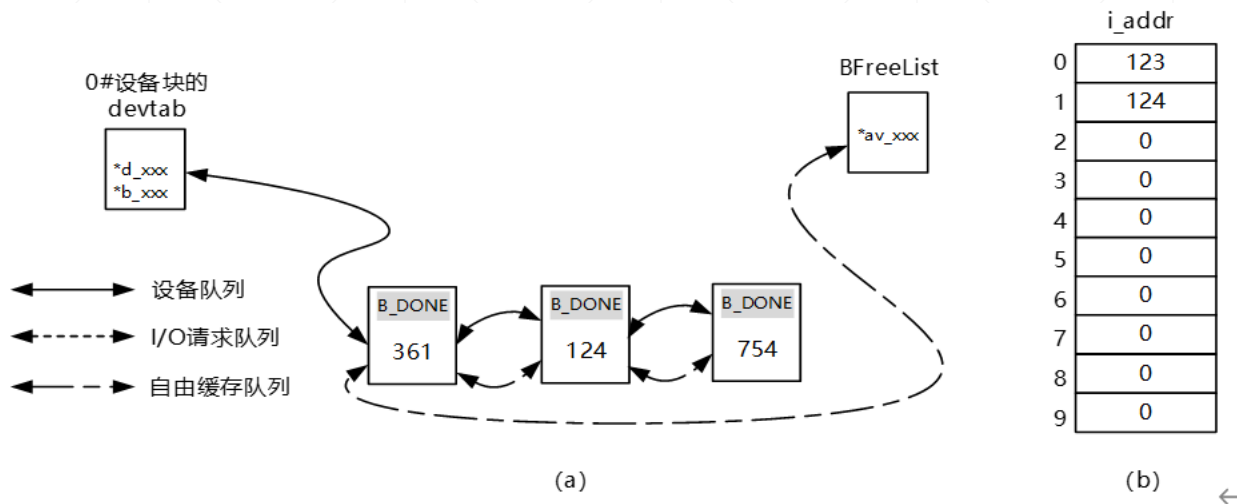


图 7.39：读写操作的初始状态

图示所有缓存自由、不脏