IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学 SHANGHAI JIAO TONG UNIVERSITY

inode文件系统

上海交通大学

https://www.sjtu.edu.cn

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

课程回顾

新指令-1：Test-and-Set

- **历史**
  - 1960年代初期，Burroughs B5000首先引入

```
1 int TestAndSet(int *old_ptr, int new) {
2     int old = *old_ptr; // fetch old value at old_ptr
3     *old_ptr = new; // store 'new' into old_ptr
4     return old; // return the old value
5 }
```

注意：TestAndSet仅为单条指令，C代码仅用于表示语义

使用 Test-and-Set 实现 Spin Lock

```
1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0 indicates that lock is available, 1 that it is held
7     lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12       ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

新指令-2：Compare-and-swap

- **另一个原子的硬件原语**
  - Compare-and-swap (on SPARC)
  - Compare-and-exchange (on x86)

```
1 int CompareAndSwap(int *ptr, int expected, int new) {
2     int actual = *ptr;
3     if (actual == expected)
4         *ptr = new;
5     return actual;
6 }
```

注意：C代码仅用于表示语义

用 Compare-and-swap 实现 Spin Lock

```
1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0 indicates that lock is available, 1 that it is held
7     lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11     while (CompareAndSwap(&lock->flag, 0, 1) == 1)
12       ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

新指令-3：Load-linked & Store-conditional

ARM架构

```
1 int LoadLinked(int *ptr) {
2     return *ptr;
3 }
4
5 int StoreConditional(int *ptr, int value) {
6     if (no one has updated *ptr since the LoadLinked to this address) {
7         *ptr = value;
8         return 1; // success!
9     } else {
10        return 0; // failed to update
11    }
12 }
```

注意：C代码仅用于表示语义

用 LL/SC 来实现 Spinlock

```
1 void lock(lock_t *lock) {
2     while (1) {
3         while (LoadLinked(&lock->flag) == 1)
4             ; // spin until it's zero
5         if (StoreConditional(&lock->flag, 1) == 1)
6             return; // if set-it-to-1 was a success: all done
7                     // otherwise: try it all over again
8     }
9 }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

新指令-4：Fetch-and-add

```
1 int FetchAndAdd(int *ptr) {
2       int old = *ptr;
3       *ptr = old + 1;
4       return old;
5 }
```

注意：C代码仅用于表示语义

用 Fetch-and-add 实现 Ticket Lock

```
1 typedef struct __lock_t {
2     int ticket;
3     int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7     lock->ticket = 0;
8     lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15 }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```

和Spin Lock相比，Ticket Lock具有公平性

偏向读者的读写锁实现示例

Reader计数器：
表示有多少读者

思考：如何实现偏向写者?

```
struct rwlock {
        int reader;
        struct lock reader_lock;
        struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
        lock(&lock->reader_lock);
        lock->reader += 1;
        if (lock->reader == 1) /* No reader there */
                lock(&lock->writer_lock);
        unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
        lock(&lock->reader_lock);
        lock->reader -= 1;
        if (lock->reader == 0) /* Is the last reader */
                unlock(&lock->writer_lock);
        unlock(&lock->reader_lock);
}

void lock_writer(struct rwlock *lock) {
        lock(&lock->writer_lock);
}

void unlock_writer(struct rwlock *lock) {
        unlock(&lock->writer_lock);
}
```

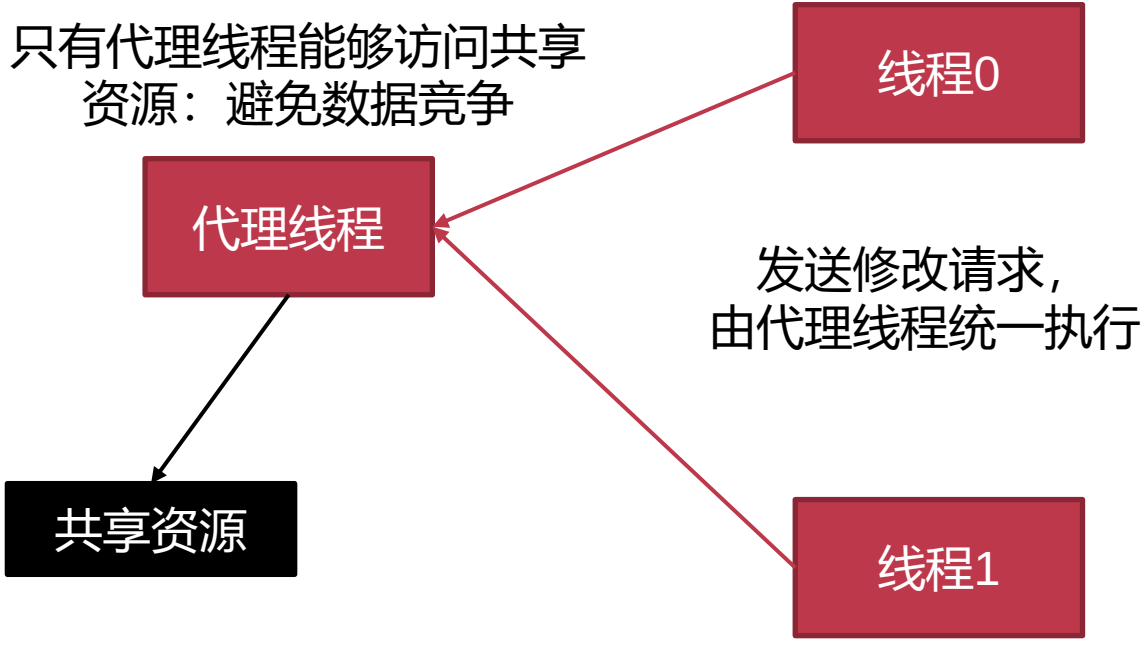
第一个/最后一个reader负责获取/释放写锁

只有当完全没有读者时，写者才能进入临界区

死锁预防：方法一

死锁条件：互斥访问

避免互斥访问：通过其他手段（如代理执行）



*代理锁 (Delegation Lock) 实现了该功能

死锁预防：方法二

死锁条件：持有并等待

不允许持有锁并等待拿其他锁：若拿不到某把锁，则把已获取的锁也放了

trylock非阻塞
立即返回成功或失败

```
while (true) {
        if(trylock(A) == SUCC)
                if(trylock(B) == SUCC) {
                        /* Critical Section */
                        unlock(B);
                        unlock(A);
                        break;
                } else
                        unlock(A);
}
```

无法获取B，那么释放A

死锁预防：方法三

死锁条件：循环等待

打破循环等待：按照特定顺序获取锁

- 对锁进行编号
- 让所有线程递增获取

A：1号 B：2号：必须先拿锁A，再拿锁B

```
void proc_A(void) {
        lock(A);
        /* Time T1 */
        lock(B);
        /* Critical Section */
        unlock(B);
        unlock(A);
}

void proc_B(void) {
        lock(B);
        /* Time T1 */
        lock(A);
        /* Critical Section */
        unlock(A);
        unlock(B);
}
```

任意时刻：获取最大资源号的线程可以继续执行，然后释放资源

死锁预防：方法四

死锁条件：资源非抢占

允许资源抢占：需要考虑如何恢复

```
void proc_A(void) {
        lock(A);
        /* Time T1 */
        lock(B);
        /* Critical Section */
        unlock(B);
        unlock(A);
}

void proc_B(void) {
        lock(B);
        /* Time T1 */
        lock(A);
        /* Critical Section */
        unlock(A);
        unlock(B);
}
```

抢占锁A

需要让执行proc_A的线程
回滚到拿锁A之前的状态

银行家算法（避免资源竞争引起的死锁）

安全序列： P2 -> P1 -> P3

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1	5	10	2	8	3	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

银行家算法：保证系统一直处于**安全状态**，且按照这个序列执行

文件系统

文件和文件系统

- **文件是对数据的一种抽象**
  - 文件的定义：有名字且持久化的一段数据

- **文件系统**
  - 提供了一组操作文件的API

回顾：文件的open/read/write操作

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    // 打开文件 hello.txt。O_RDONLY 表示以只读方式打开
    int fd = open("hello.txt", O_RDONLY);

    char result[14];
    // 从 fd 对应的文件中读取 13 个字节
    read(fd, result, 13);
    // 向屏幕输出读取到的字符串
    write(1, result, 13);
    close(fd);
}
```

UNIX文

- OPEN, RE
- FSYNC
- STAT, CH
- RENAME,
- MKDIR, C
- MOUNT, U
- ....

写一个C语言的文件读写示例：使用open、read、write、close。代码高亮

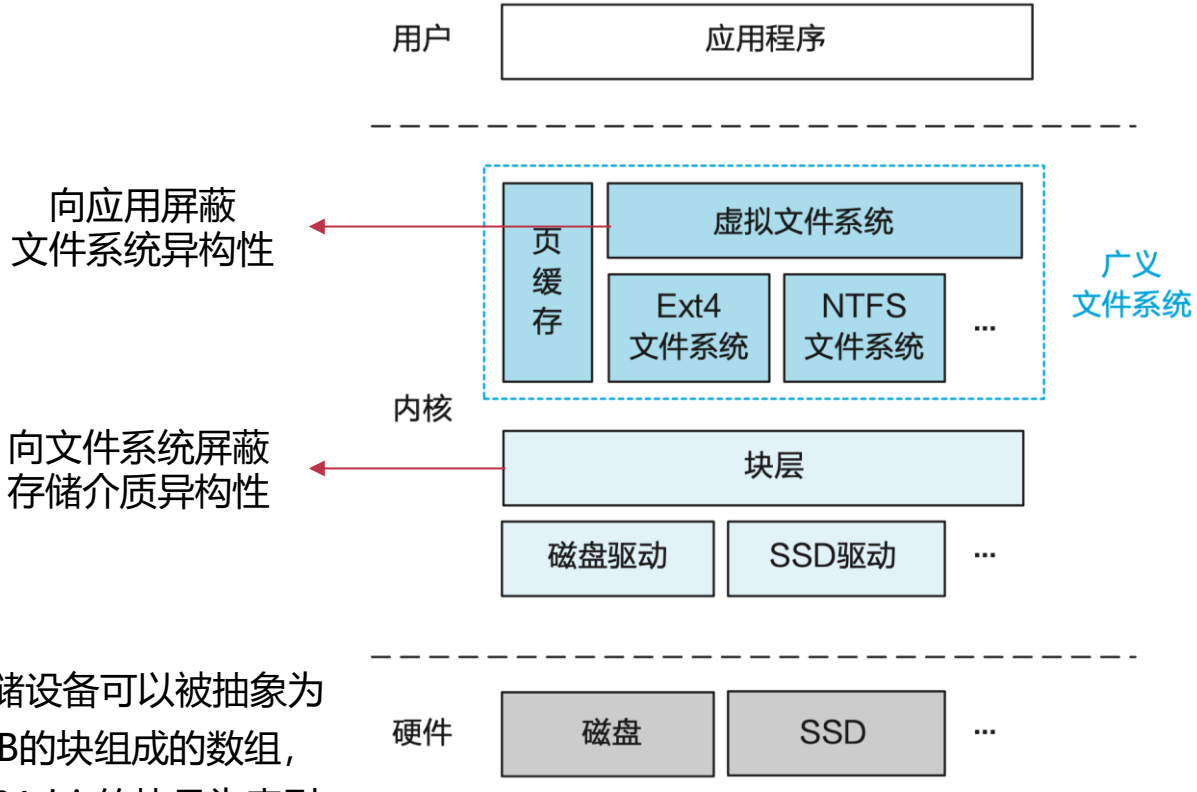
下面是使用 C 语言的 `open`、`read`、`write`、`close` 函数实现文件读写的示例：

Copy code

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
  // 打开文件
  int fd = open("test.txt", O_RDWR | O_CREAT, 0644);
  if (fd < 0) {
    // 文件打开失败
    perror("Error opening file");
    return 1;
  }

  // 写入文件
  const char *text = "This is a test\n";
  write(fd, text, sizeof(text) - 1);

  // 读取文件
  char buf[100];
  lseek(fd, 0, SEEK_SET); // 移动文件指针到文件开头
  read(fd, buf, sizeof(buf));
  printf("%s", buf);

  // 关闭文件
  close(fd);

  return 0;
}
```

性介质

文件系统的位置



OPEN("a.txt", "rw")

READ(…)

WRITE(…)

…

READ(block-addr, buf)

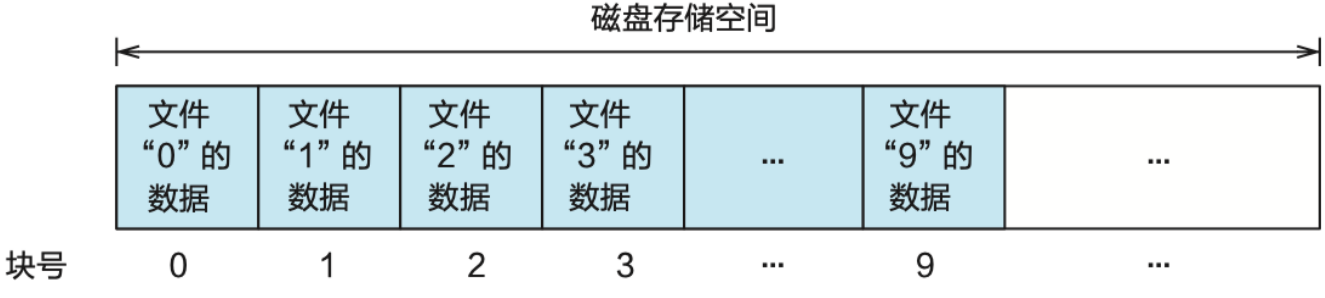
WRITE(block-addr, buf)

存储设备可以被抽象为
4KB的块组成的数组，
以64-bit的块号为索引

INODE：文件的元数据

一种DIY的简单文件系统

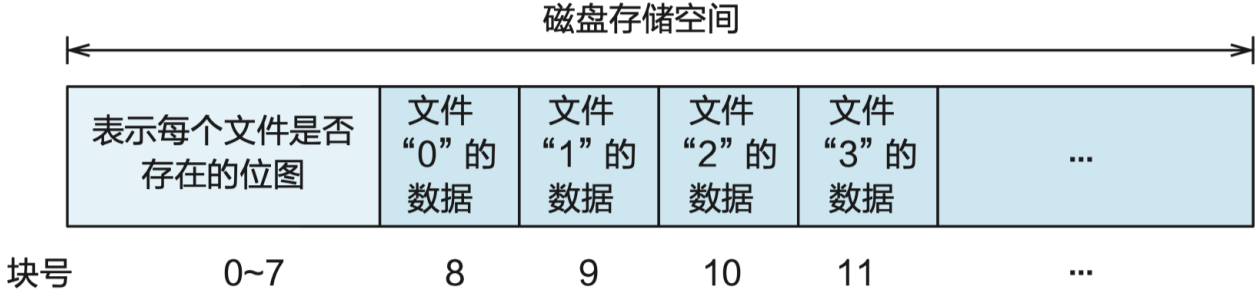
- **文件系统的特性**
  - 文件数量固定：10 个文件
  - 文件名固定：从 “0”到 “9”
  - 每个文件的大小固定为 4 KB
  - 10个文件在磁盘上是连续的
- **文件系统的操作**
  - 文件查找：文件名就是磁盘块号
  - 文件读写：即对相应磁盘块的读写
  - 文件删除：不支持



把磁盘抽象为一个大数组，块号就是磁盘的索引，每个块大小为4KB

简单文件系统的改进

- **增加文件的数量**
  - 假设使用容量为 1 GB 的磁盘，最多保存 1GB / 4 KB = 256K 个文件
- **支持文件删除操作**
  - 引入文件位图（bitmap），大小为256K × 1b = 8 × 4KB
  - 每个bit对应一个文件，若bit为0表示不存在，bit为1表示存在
  - 删除文件时，将该文件对应的bit设置为0

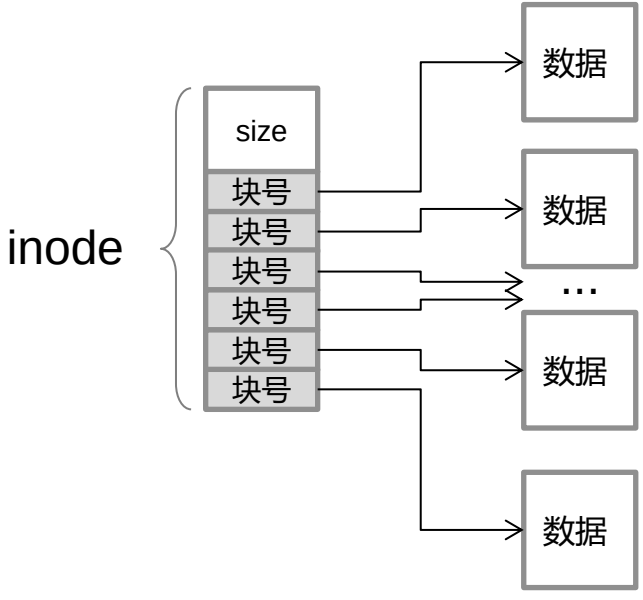


简单文件系统的限制

- **文件大小固定**
  - 无法支持大于一个磁盘块（4KB）的文件
  - 文件系统依赖文件数据存储的连续性
- **文件名固定**
  - 只能用磁盘块号来表示文件名

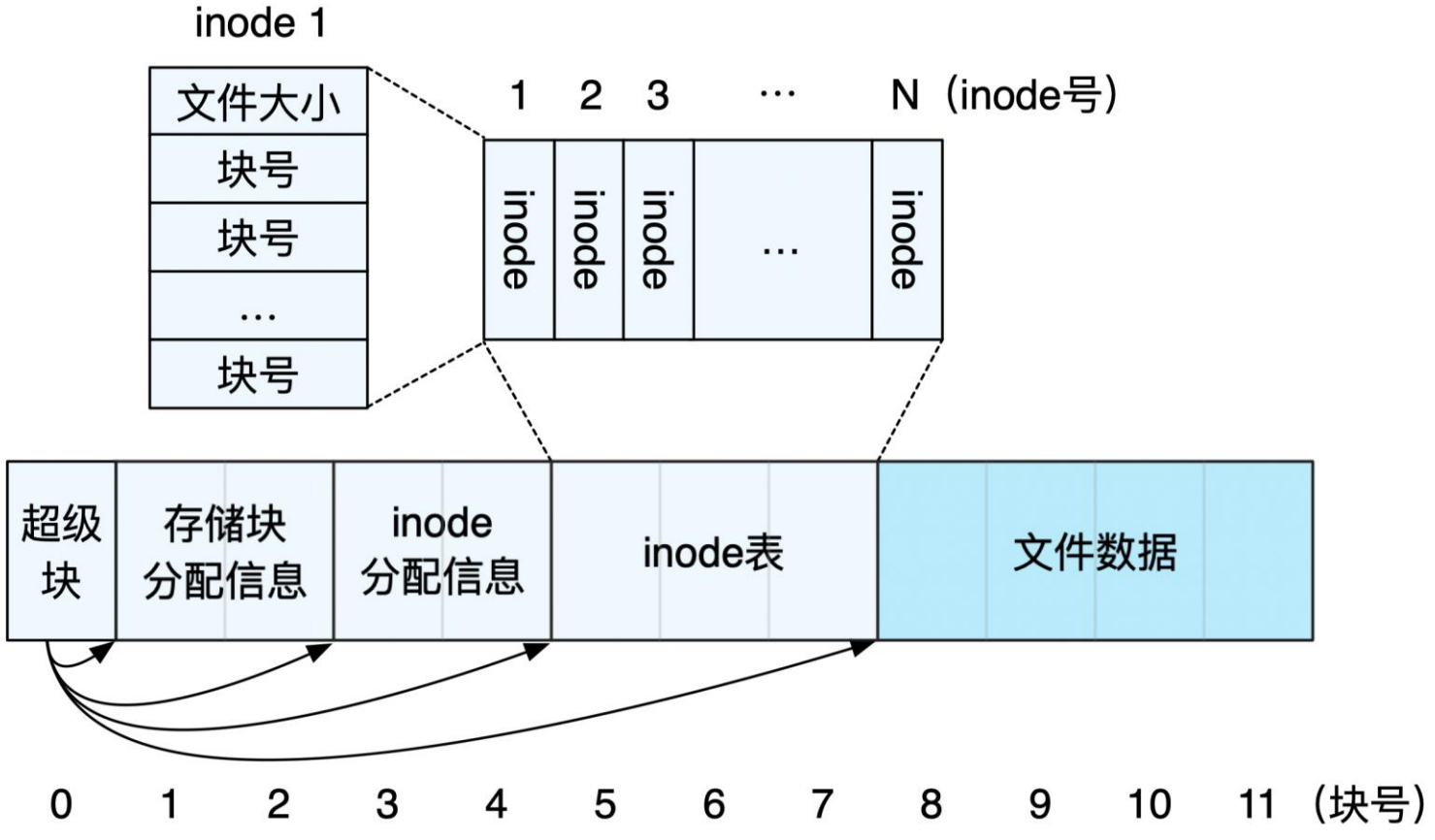
inode：记录文件多个磁盘块的位置

- **引入inode：index node**
  - 记录多个磁盘块号
  - 头部记录文件size信息
  - 每个文件对应一个inode
  - 称为文件**元数据**（Metadata）
- **文件读写操作**
  - 给定inode和文件内偏移（offset）
  - 根据offset计算出对应的磁盘块号
  - 若offset超出size则返回错误



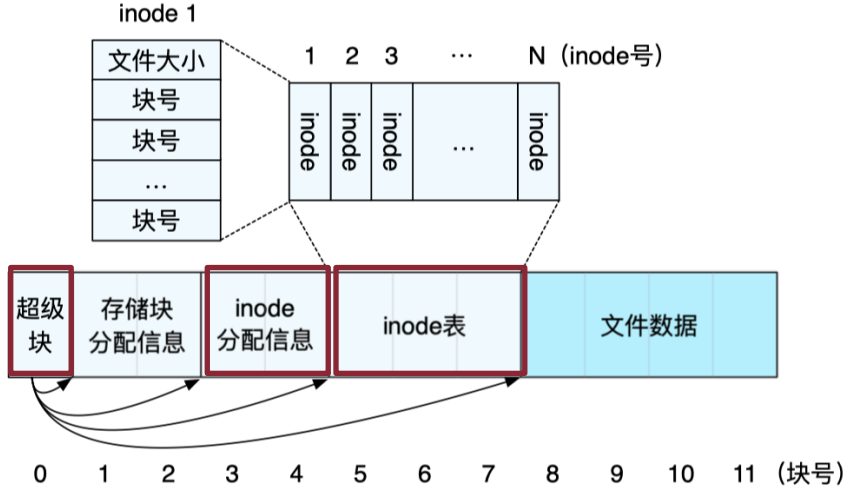
最简inode，后面会扩展

inode文件系统的存储布局

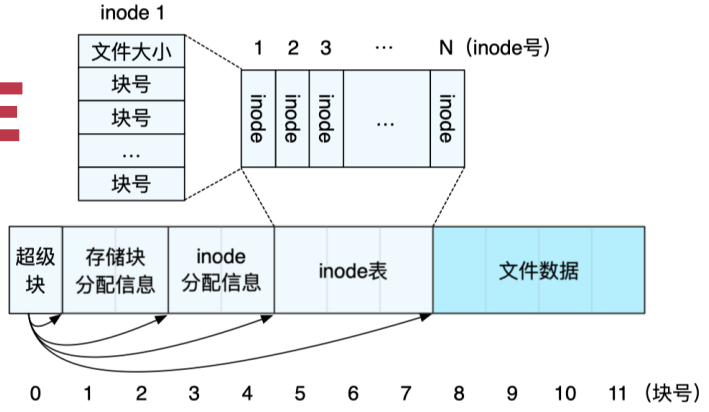


inode文件系统的存储布局

- **inode表：记录所有inode**
  - 可以看成inode的大数组
  - 每个inode使用作为索引
  - 此时，inode号即为文件名
- **inode分配信息（位图）**
  - 记录哪些inode已分配，哪些空闲
- **超级块：Super Block**
  - 记录磁盘块的大小、其他信息的起始磁盘块位置，等等
  - 是整个文件系统的元数据



inode文件系统的基本操作



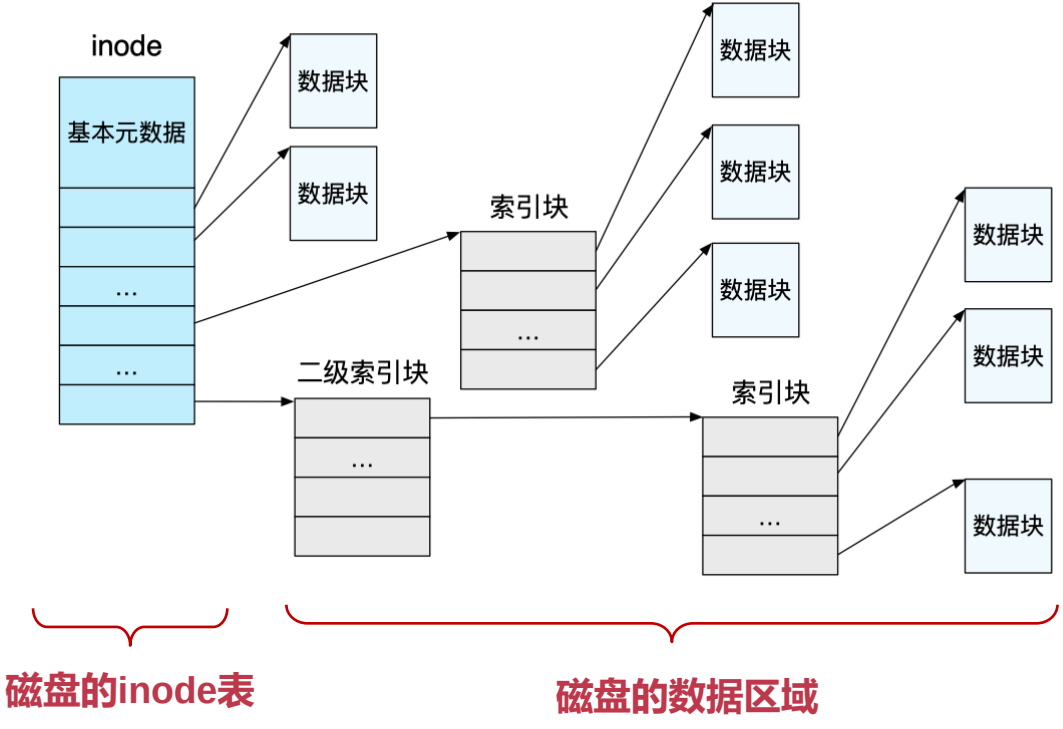
- **加载文件系统**
  - 首先读取超级块，然后找到其他信息
- **创建新文件**
  - 根据inode分配信息找到空闲inode，将inode对应的bit设置为1
  - 返回inode在inode表中的索引，作为文件名
- **查找文件（根据inode号）**
  - 在inode表中根据inode号定位该inode
- **删除文件**
  - 在inode分配表中，将该inode对应的bit设置为0

单级inode过大的问题

- **一个4GB的文件，对应inode有多大?**
  - 假设磁盘块号（块指针）为8-Byte（64-bit）
  - inode大小：4GB/4KB * 8 = 8MB
  - 若文件大小为4TB，则inode大小为8GB!
- **inode表的假设**
  - 单个inode在磁盘上的空间是连续的
    - 一个inode记录的所有块指针在磁盘上的空间是连续的
  - 多个inode在磁盘上的空间是连续的
  - **单级inode会导致预留的inode表过大**

多级inode

- 引入**索引块**：指向数据块；以及**二级索引块**：指向索引块；...
- 索引块（包括二级索引块）不在inode表的存储区域，而是在数据区域

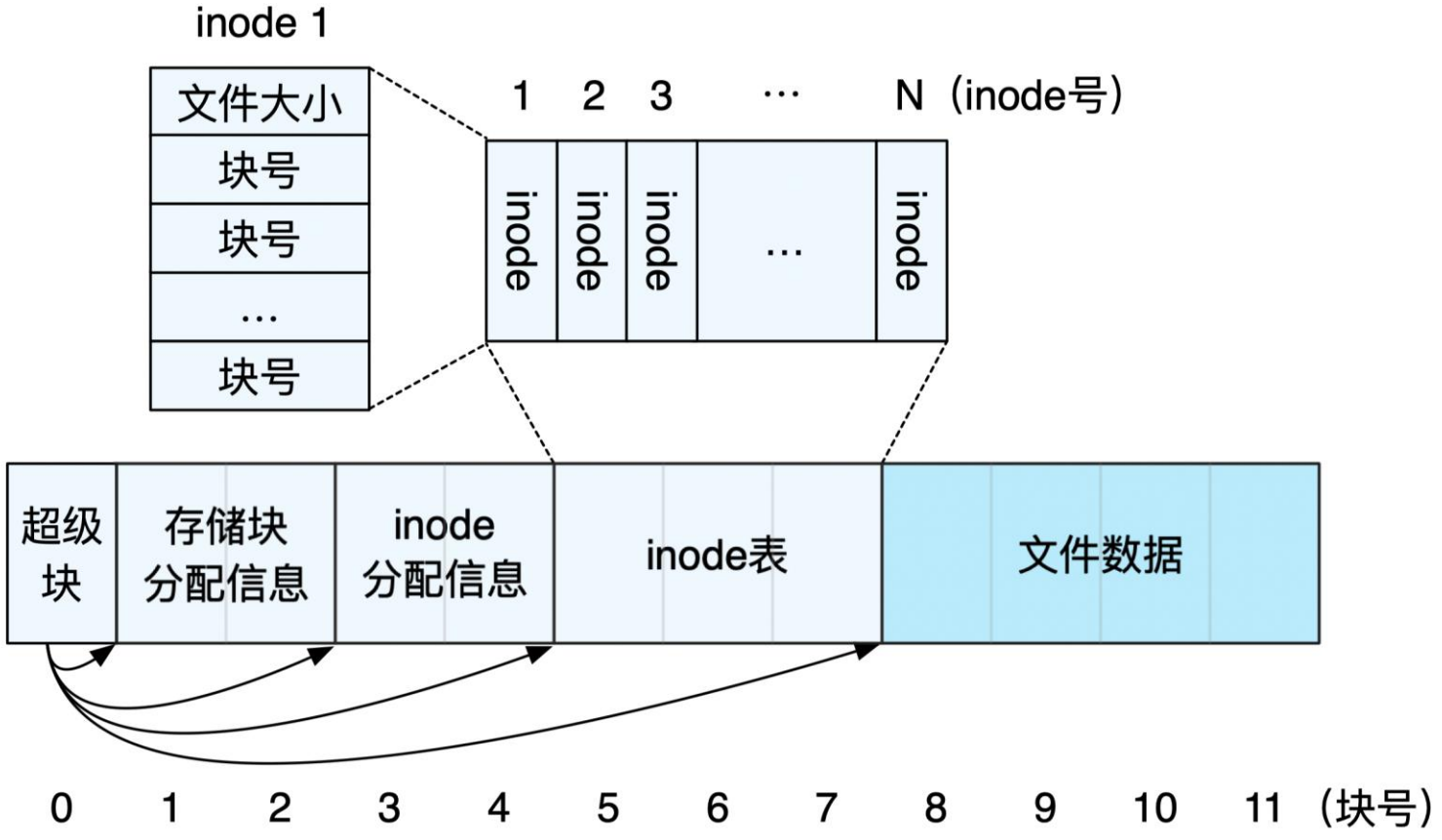


多级inode

- **一个多级inode占用的空间很少**
  - 一共只有16个指针（即记录磁盘块），这些指针占用128-Byte
  - 包含12个直接指针，3个间接指针，1个二级间接指针
  - 文件最大为：4K x 12 + 4K x 512 x 3 + 4K x 512 x 512 = 48K + 6M + 1G
- **如何支持更大的文件?**
  - 可以启用三级索引，甚至四级索引

问：课程中上一个用多级数据结构减小占用空间的机制是?

问：为什么格式化后可用空间变小了？



目录：也是一种文件

inode与文件名

- **inode本身已经包含了一个文件的所有信息**
  - 可以使用inode号（inode表的索引）作为文件名
  - 给定一个inode号，就可以访问文件的所有数据
- **inode作为文件名的缺点**
  - 名字很难记住，不够 user-friendly
  - 名字依赖于inode表位置的名字
    - 一旦改变了位置，就必须改变文件名（从U盘拷贝进电脑）

用字符串做文件名

Q：文件名属于文件吗?

- **以字符串作为文件名的好处**
  - 在操作文件时，将文件的元数据隐藏起来，用户无需感知
  - 不依赖特定的存储设备

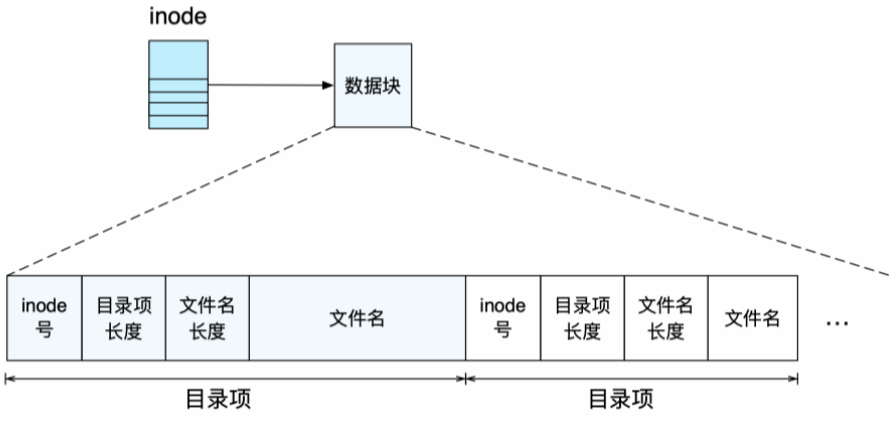
- **如何实现字符串文件名到inode号的映射?**
  - 使用映射表，记录**字符串到inode号的映射**
  - 将该表保存在一类特殊的文件中，称为目录文件
    - 目录本身也是一个文件，同样有inode
    - 复用inode机制来实现目录!
    - **inode扩展：增加类型**（区分普通文件和目录）

File name	Inode number
program	10
paper	12

目录的内容：与书的目录类似

目录文件与目录项

- **目录中的每条映射称为一个目录项**
  - 每一条目录项记录了一个inode号与文件名字符串的映射
  - 一个目录可以记录很多目录项
- **目录文件的大小（占用空间）**
  - 与其记录的文件大小无关
    - 思考：与什么因素有关?
- **目录支持查找操作**
  - 给定一个目录文件和字符串
  - 在目录文件中查找字符串，并返回对应的inode



思考：文件夹大小

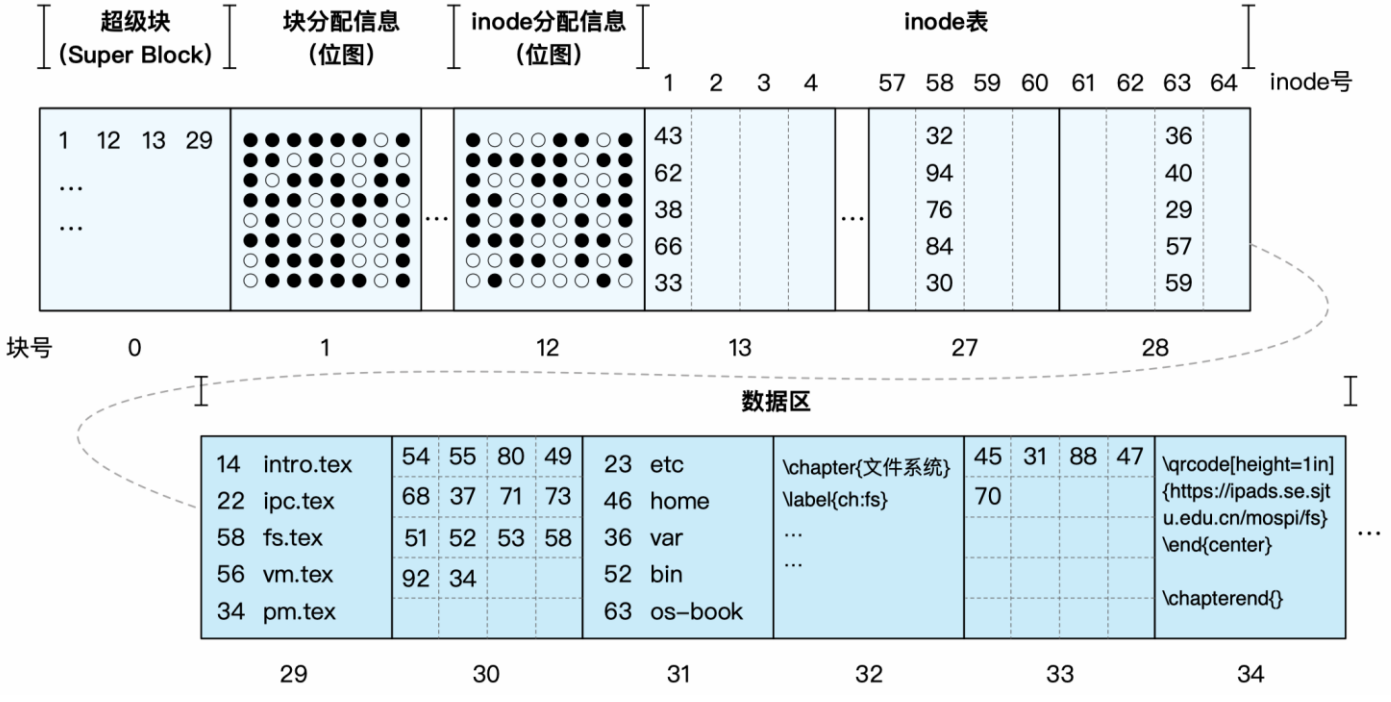
- 文件夹1：10个小文件
- 文件夹2：1个大电影
- 问：请问哪个文件夹大?
- **答：无法判断（通常来说是文件夹1大）**

“文件夹”具有一定的误导性，“目录”命名更合理。
文件夹/目录大小与存储的文件大小无关。

目录的递归与根目录

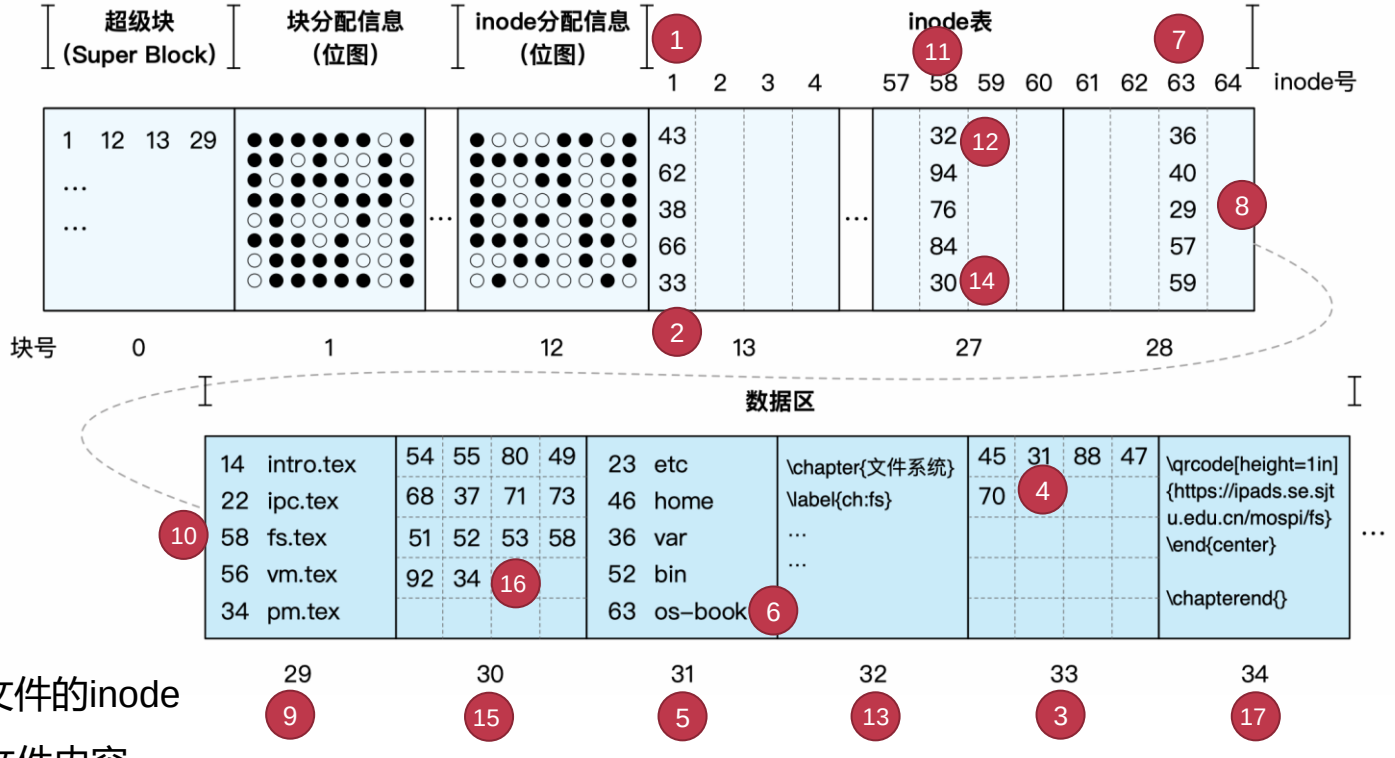
- **目录中可以记录子目录**
  - 因为目录本身也是一个文件
  - 通过“/”来分割父目录和子目录
- **最顶端的目录没有目录名（文件名）**
  - 被称为“根目录”（root）
  - 根目录没有文件名，在“/”的前面什么都没有
  - 通常固定为1号inode
- **绝对路径和相对路径**
  - 绝对路径：如“/home/OS/test.md”
  - 相对路径：如“./test.md”或“OS/test.md”

文件的查找过程：/os-book/fs.tex



注意：inode在这里并没有size等其他信息，只记录了block number（**最后一个指向索引块**）；**根目录是1号inode**；块分配信息是从29号块开始计算，即只考虑数据块；没有画出来的block可忽略。

文件的查找过程：/os-book/fs.tex

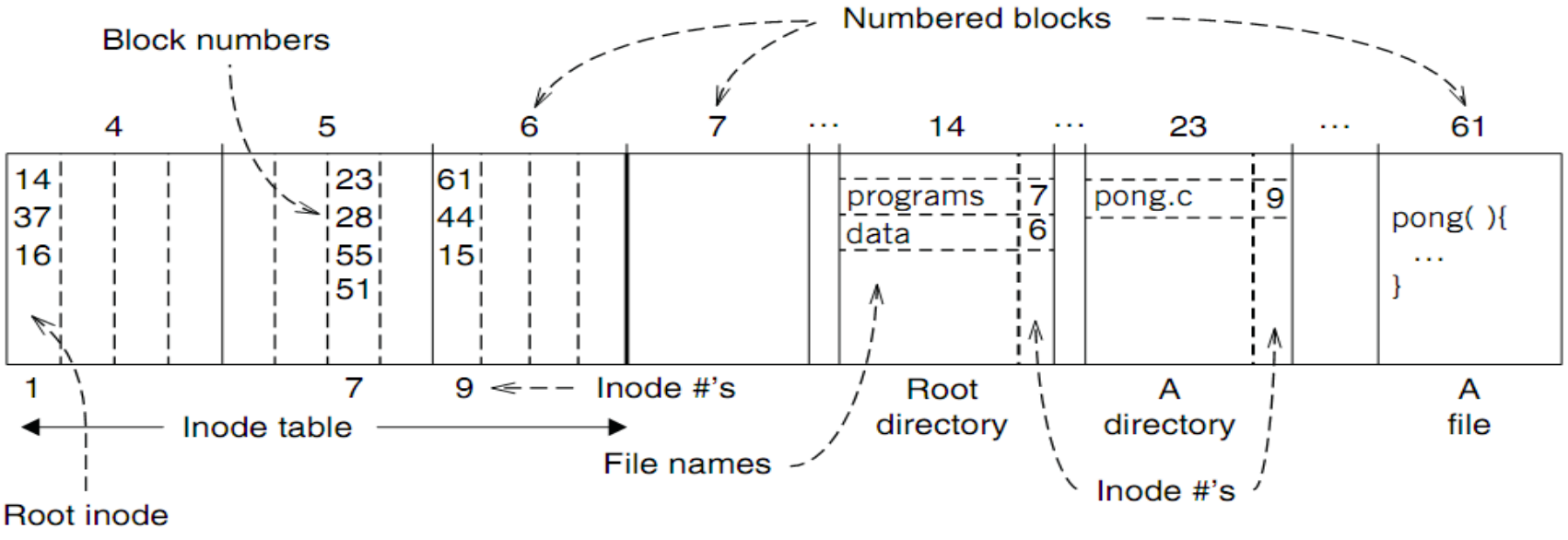


1~11 找到文件的inode

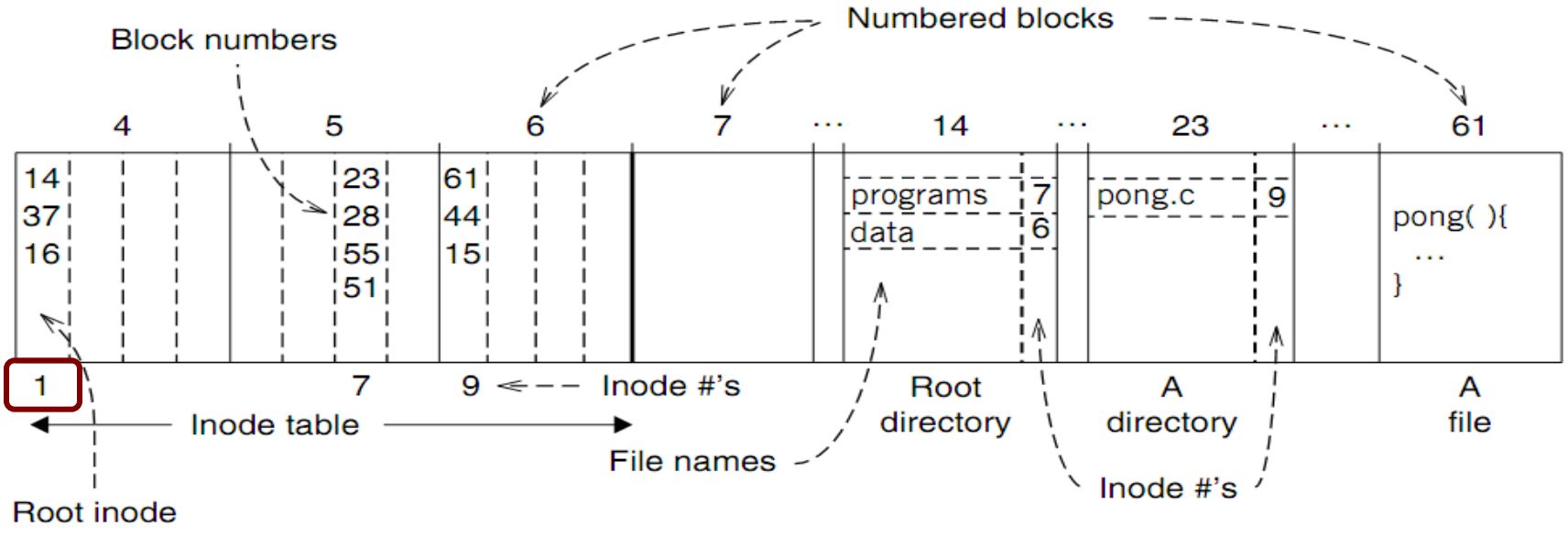
12~17 读取文件内容

注意：inode在这里并没有size等其他信息，只记录了block number（最后一个指向索引块）；根目录是1号inode；块分配信息是从29号块开始计算，即只考虑数据块；没有画出来的block可忽略。

练习：根据文件名找到文件块："/programs/pong.c"

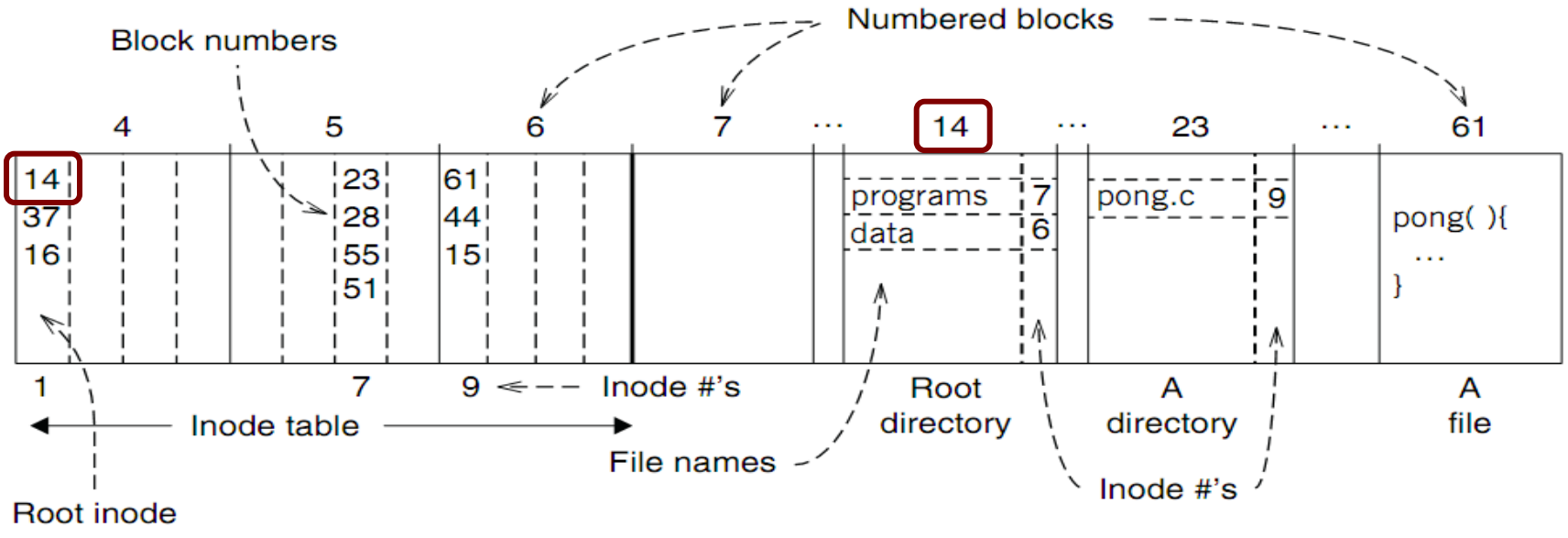


练习：根据文件名找到文件块："/programs/pong.c"



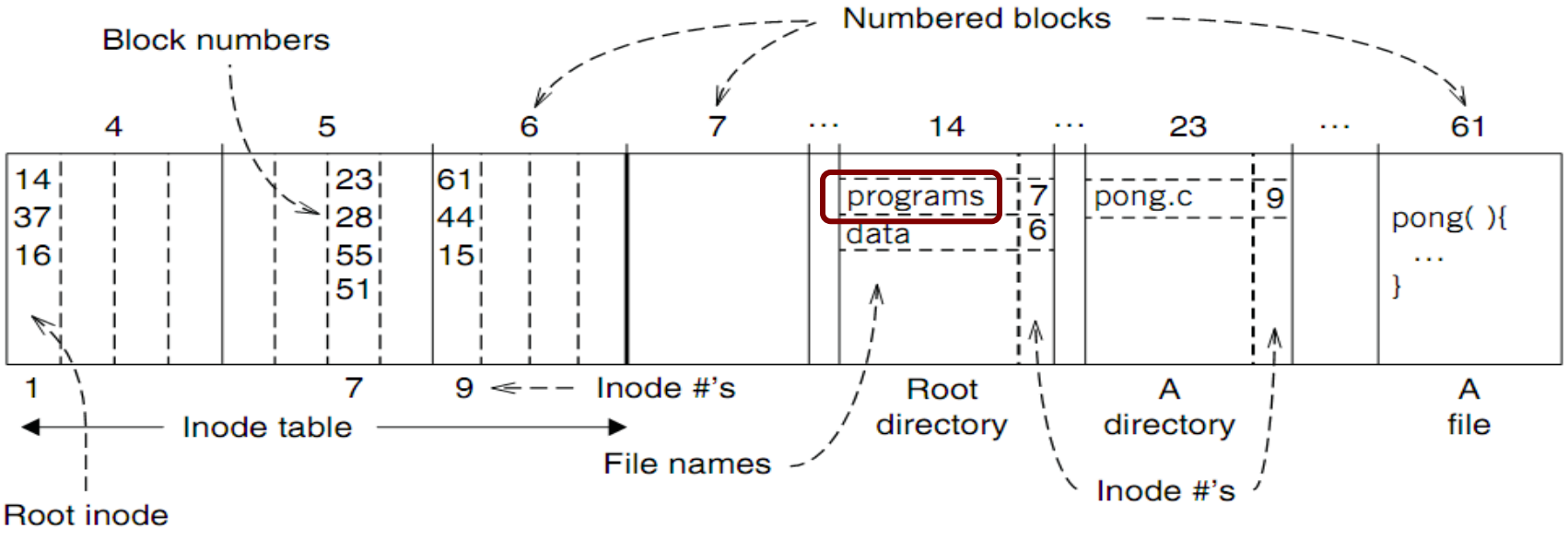
- 找到 '/' 根目录的 inode: 1

练习：根据文件名找到文件块："/programs/pong.c"



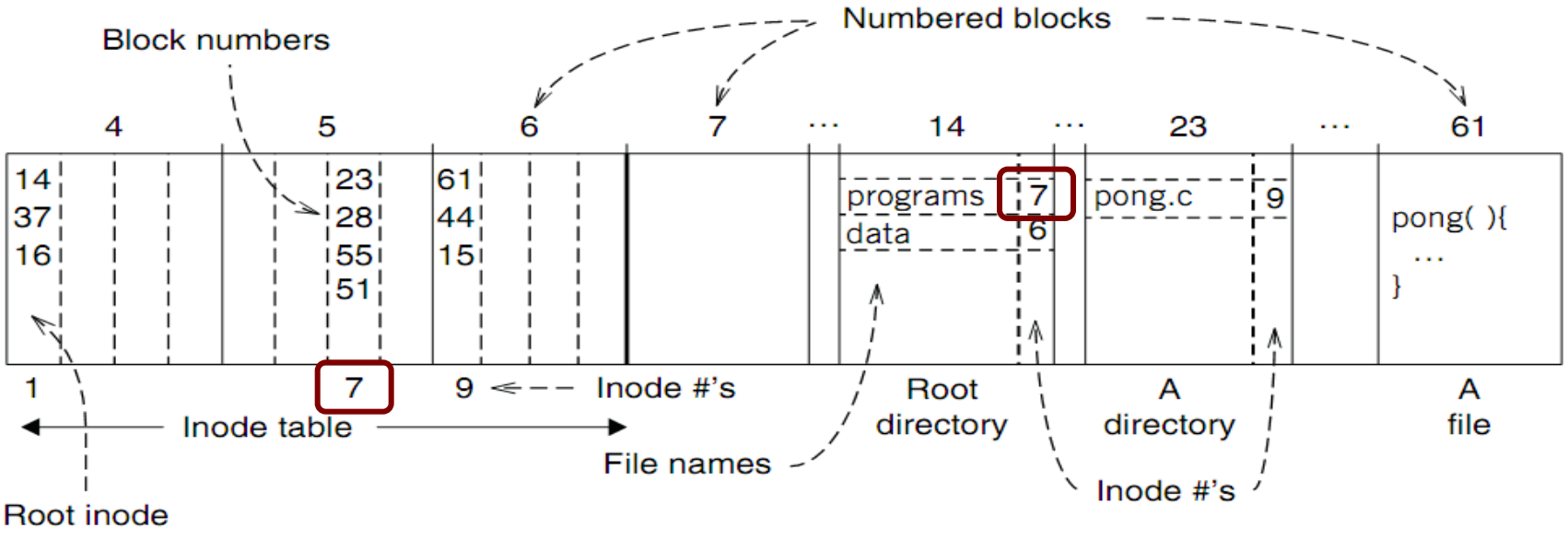
- **找到根目录文件的数据块：14号磁盘块**

练习：根据文件名找到文件块："/programs/pong.c"



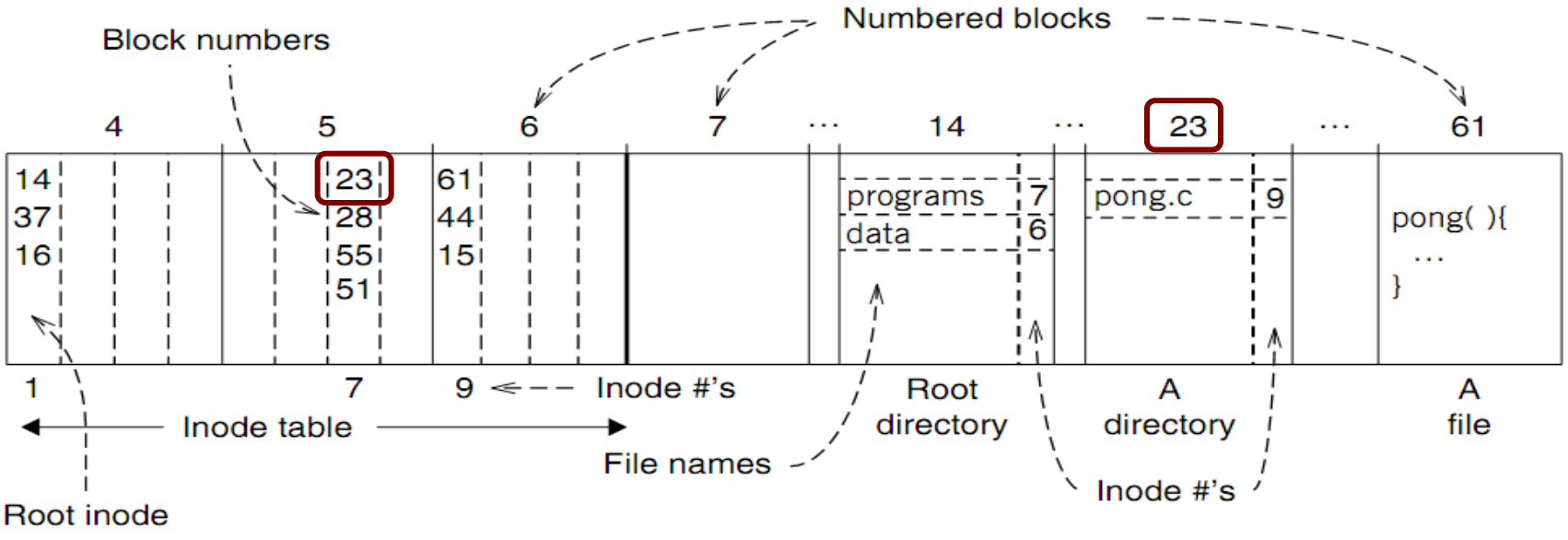
- 在根目录中，通过字符串比较，找到 '/programs' 对应的目录项

练习：根据文件名找到文件块："/programs/pong.c"



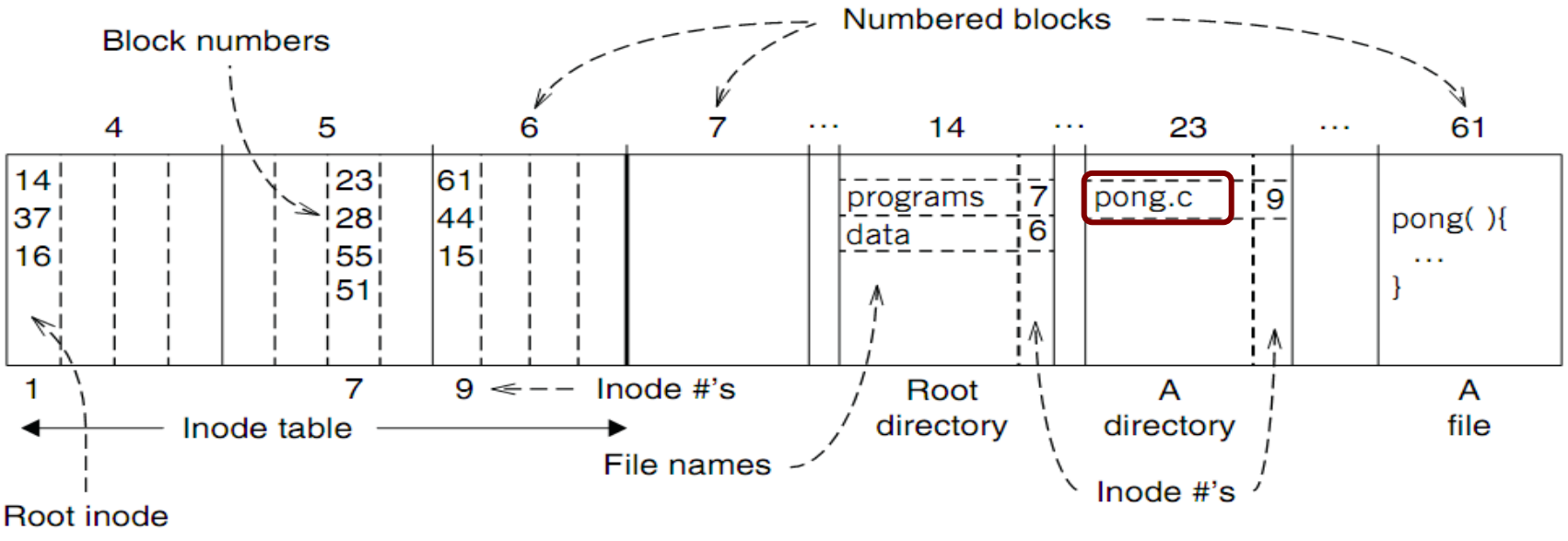
- 根据 inode号（7）找到 '/programs' 的inode

练习：根据文件名找到文件块："/programs/pong.c"



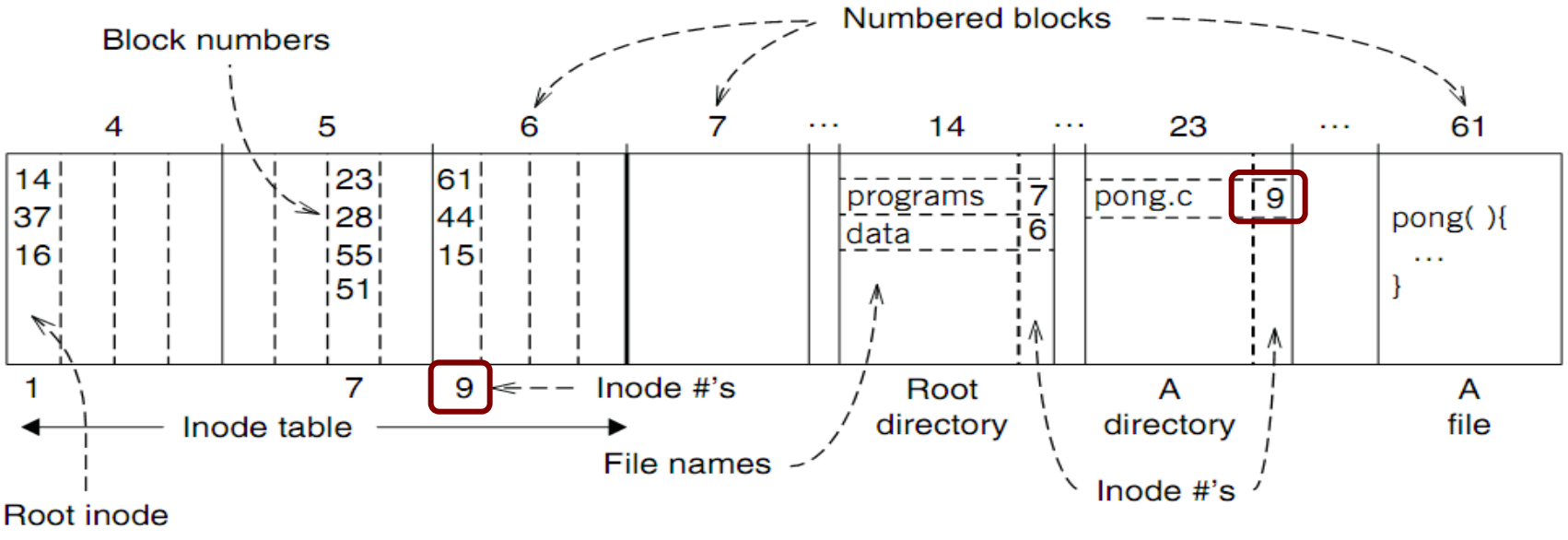
- 找到 programs 目录文件的磁盘块：23号磁盘

练习：根据文件名找到文件块："/programs/pong.c"



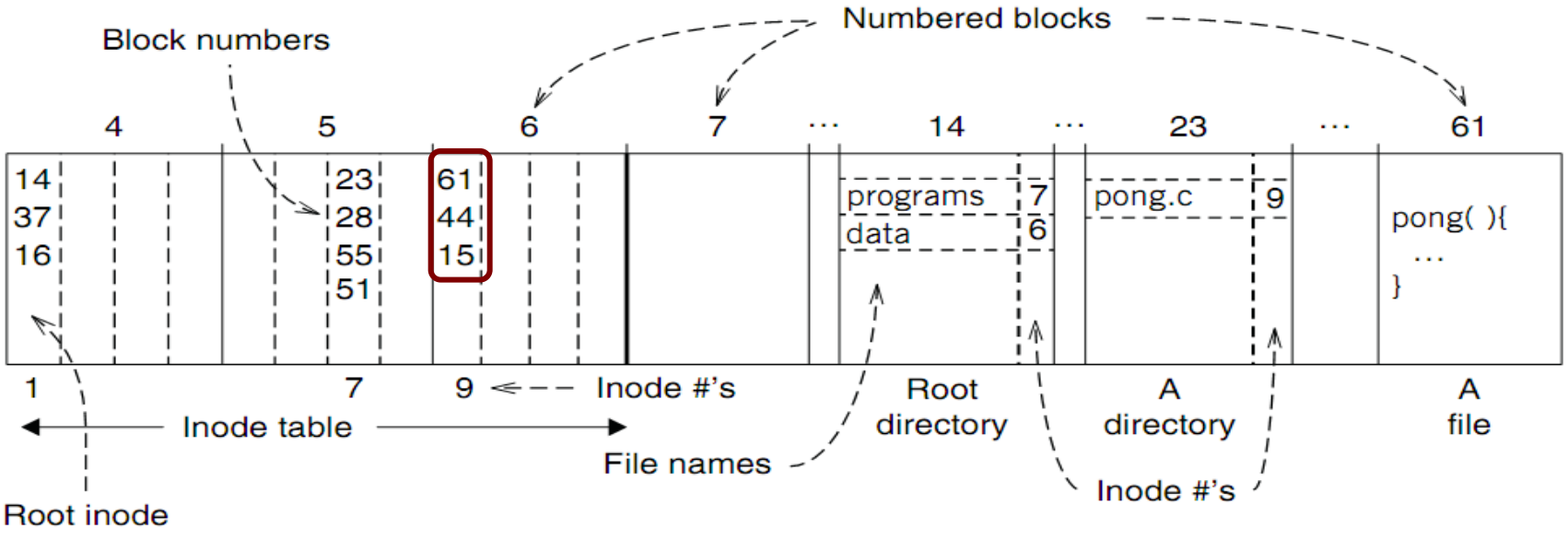
- 通过比较字符串，在目录中找到 '/programs/pong.c' 的目录项

练习：根据文件名找到文件块："/programs/pong.c"



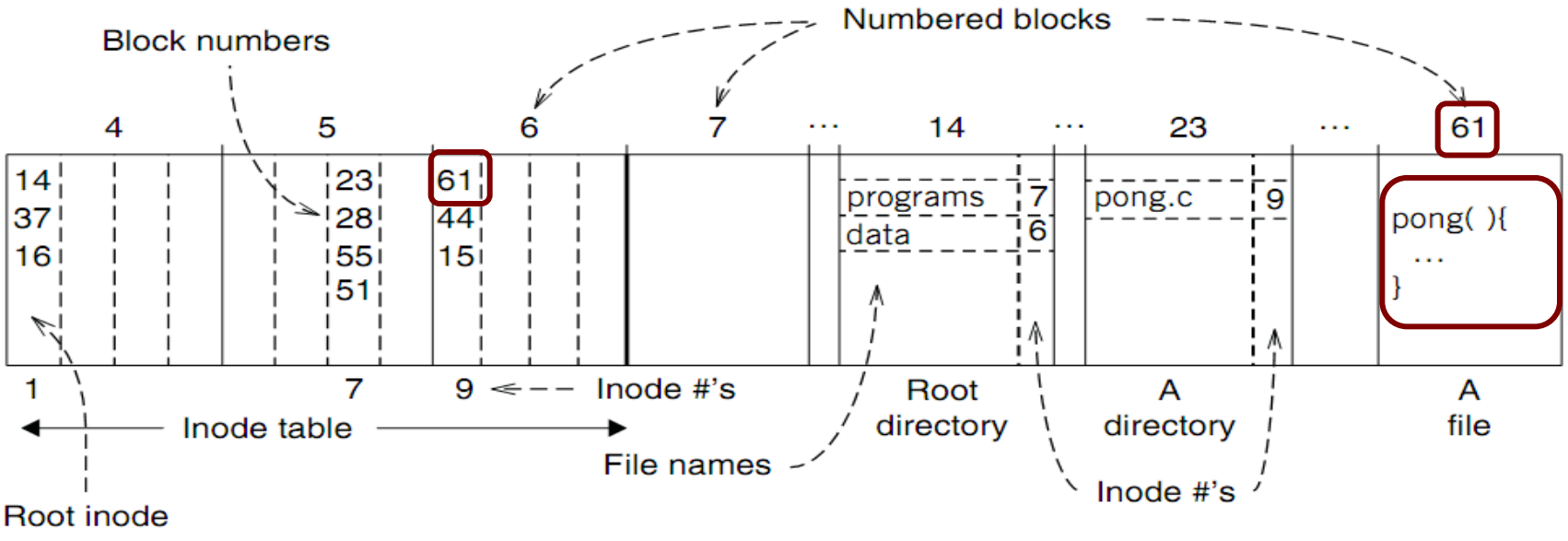
- 根据 inode号（9）找到 '/programs/pong.c' 的inode

练习：根据文件名找到文件块："/programs/pong.c"



- 找到 '/programs/pong.c' 的数据存放在文件块61、44和15号数据块

练习：根据文件名找到文件块："/programs/pong.c"



- 找到 61号块的数据，以及44和15号块数据，即为文件内容

硬链接与软链接

创建（硬）链接：Linux中的 ln 命令

```
[user@osbook ~] $ touch a
[user@osbook ~] $ ln a b
[user@osbook ~] $ echo "hello, world" > b
[user@osbook ~] $ cat a
hello, world
[user@osbook ~] $ ls -i
90050313 a    90050313 b
[user@osbook ~] $ mkdir d
[user@osbook ~] $ ln d e
ln: d: Is a directory
```

（硬）链接：Link

- **LINK**
  - LINK("Mail/inbox/new-assignment", "assignment")
  - 将严格的层次结构（树）变成有向图
    - 注意：用户不能为目录创建link
  - 不同的文件名可以指向同一个inode号
- **UNLINK**
  - 删掉从文件名到inode号的绑定关系
  - 如果 UNLINK 最后一个绑定，则把 inode 和对应的 blocks放到 free-list
    - **每个文件都需要一个 reference counter**

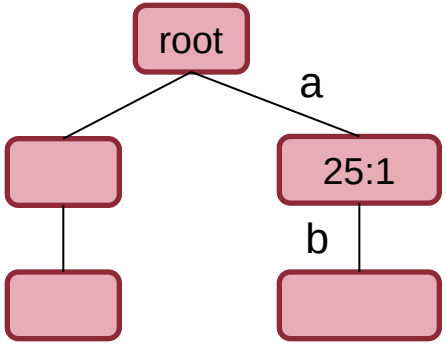
（硬）链接：Link

- **引用计数器（Reference count）**
  - 一个 inode 可以绑定多个文件名
    - LINK 时 +1, UNLINK 时 -1
  - 当reference count为0时，文件被删除
  - 不允许出现环
    - 除了 '.' 和 '..'
    - 用来表明当前目录和上一层目录而不需要知道它们实际的名字

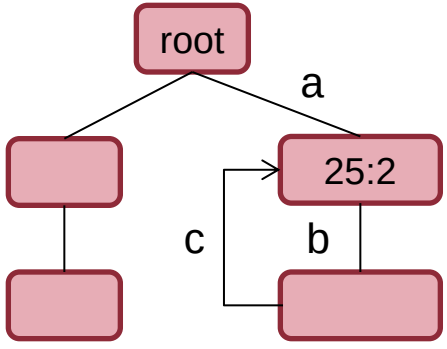
```
structure inode
    integer block_numbers[N]
    integer size
    integer type
    integer refcnt
```

inode扩展：需包含refcnt

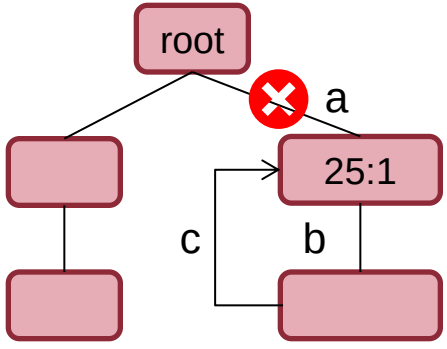
Link不能形成环



- /a/b 是一个目录
- a 的 refcnt 是1
- a 的 inode 号是25



- LINK ("/a/b/c", a")
- 造成一个环
- a 的 refcnt 是2



- UNLINK ("/a")
- a 的 refcnt 降为1（25号 inode 没有被删除 ）
- 但现在无法访问25号 inode!

软链接（符号链接）

- **如何在一个磁盘上建立指向另一个磁盘的Link?**
  - 不行，因为不同磁盘的 inode 命名空间是不同的（文件系统不同）

- **软链接（符号链接）** **soft link (symbolic link)**
  - SYMLINK
  - 增加一种新的 inode 类型

创建软链接：Linux中的 ln -s 命令

```
[user@osbook ~] $ ln -s a a.link
[user@osbook ~] $ cat a.link
cat: a.link: No such file or directory
[user@osbook ~] $ echo "hello, world" > a
[user@osbook ~] $ cat a.link
hello, world
[user@osbook ~] $ ls -l
-rw-r--r--  1 osbook  wheel  13  6 12 19:45 a
lrwxr-xr-x  1 osbook  wheel   1  6 12 19:45 a.link -> a
[user@osbook ~] $ ls -i
90055602 a      90055547 a.link
[user@osbook ~] $ mkdir d
[user@osbook ~] $ ln -s d d.link
[user@osbook ~] $ touch d/x
[user@osbook ~] $ ls d.link
x
[user@osbook ~] $ ln -s e e
[user@osbook ~] $ cd e
cd: too many levels of symbolic links: e
```

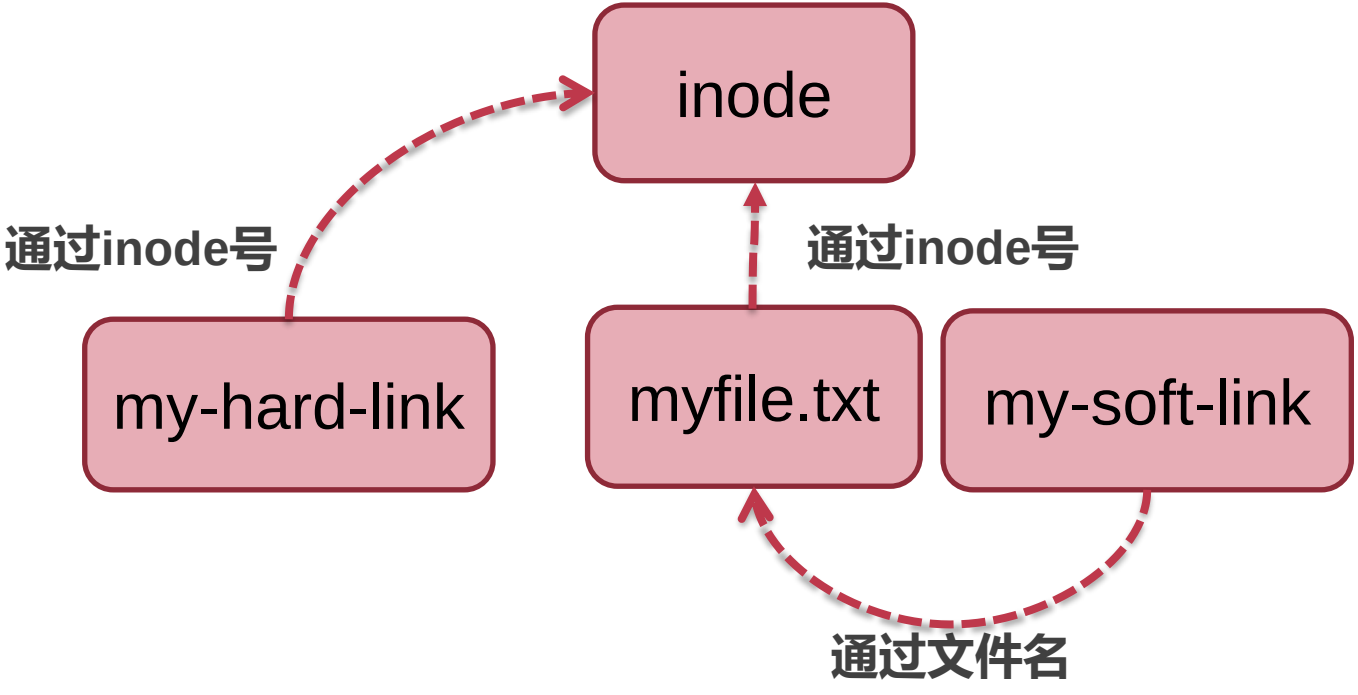
软链接可以无中生有

可以建立目录的软链接

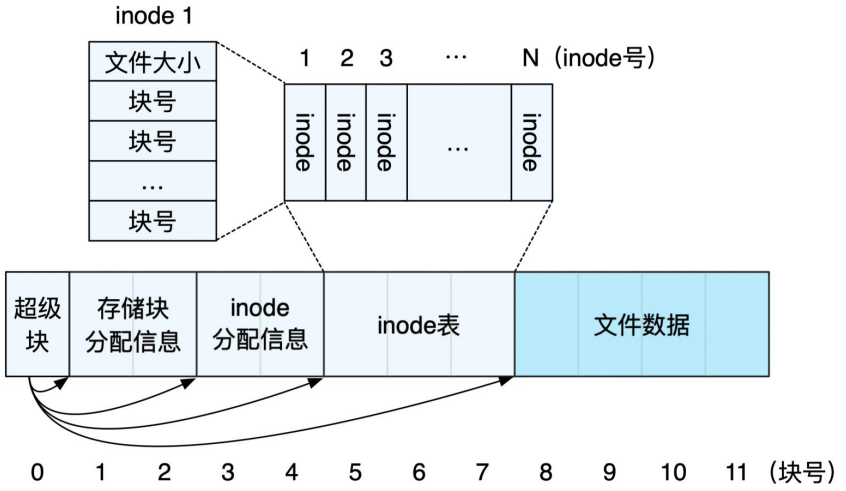
练习

```
tmac@intel12-pc:~/ieee-ai-os/file$ touch a
tmac@intel12-pc:~/ieee-ai-os/file$ ln a b
tmac@intel12-pc:~/ieee-ai-os/file$ echo "Hello World" > b
tmac@intel12-pc:~/ieee-ai-os/file$ ln -s a c.soft
tmac@intel12-pc:~/ieee-ai-os/file$ rm a
tmac@intel12-pc:~/ieee-ai-os/file$ cat c.soft

tmac@intel12-pc:~/ieee-ai-os/file$ cat b

tmac@intel12-pc:~/ieee-ai-os/file$
```

硬链接和软链接的对比


inode
通过inode号
通过inode号
my-hard-link
myfile.txt
my-soft-link
通过文件名

小结



- **存储设备上文件系统的组织**
- **inode**
  - 普通文件
    - 文件名不是文件的数据，也不是文件的元数据（inode）
  - 目录
    - 文件名是目录的数据
    - 目录所占磁盘空间通常是很小的，负责记录文件名到inode号的映射
  - 链接
    - 硬链接：一个inode可以有多个文件名（. 和 ..）注意不是新的inode 类型
    - 软链接：一种特殊的inode（快捷方式）