中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

文件系统II

SSE202/204: 操作系统原理

苏玉鑫
suyx35@mail.sysu.edu.cn
助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

大纲

- 块设备
- 基于inode的文件系统
- 基于表的文件系统
  - FAT
  - NTFS
- 使用文件系统
  - 以命令行的形式使用
  - 在代码里使用
    - 基本操作
    - 内存映射
  - 文件系统的高级功能
- 虚拟文件系统
- 用户态文件系统

硬链接与符号链接

- inode编号是给操作系统看的，而文件名是给人看的

- 文件名是存在目录里的

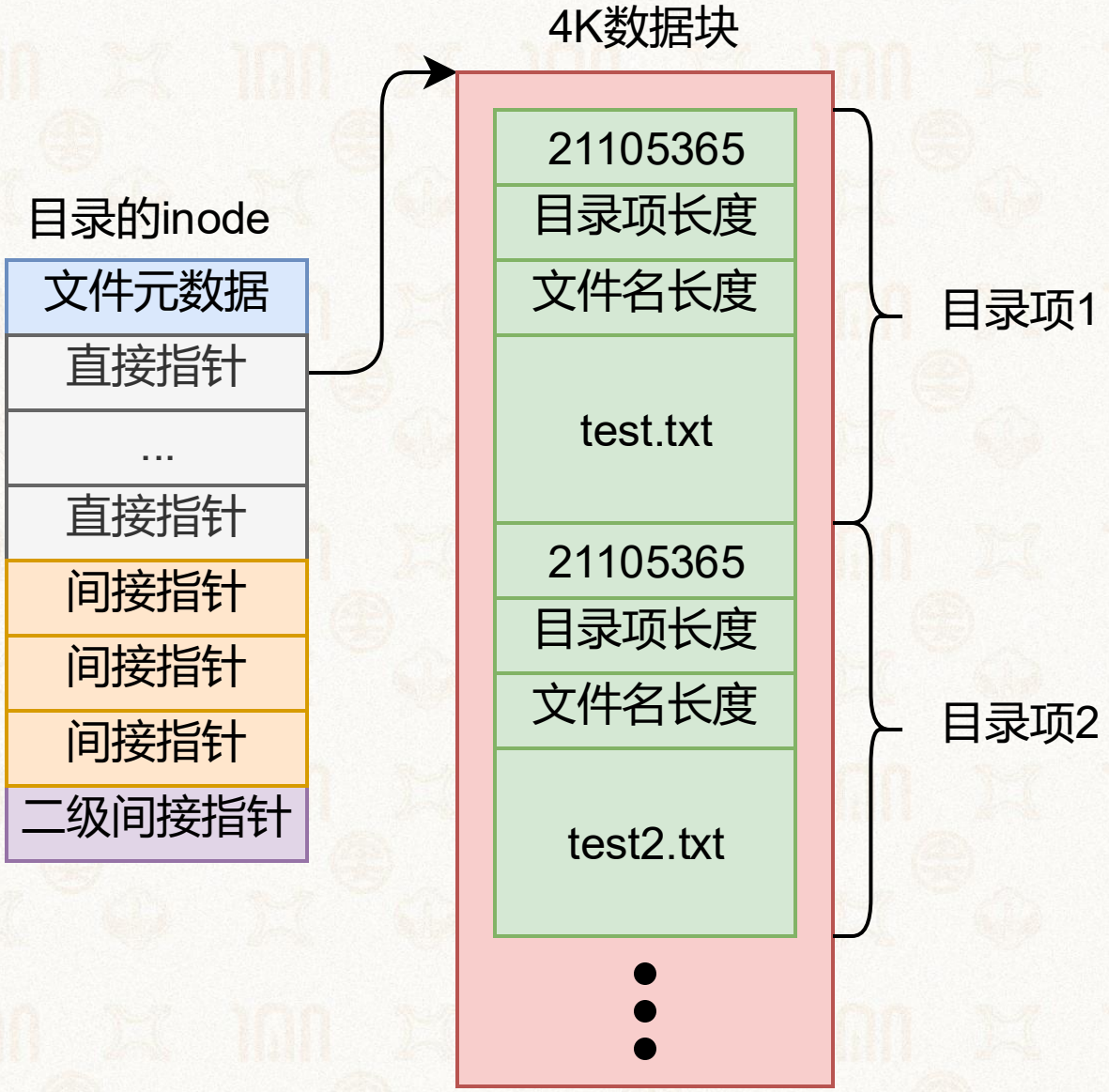
```
yxsu@Dell-T6401:~/os$ ls -i process/
21105362 a.out                  21105368 fork_demo          21105367 fork_readfile.c
21105366 myecho                 21105365 test.txt           21105373 vfork_demo.c
21105364 execve_demo            21105363 fork_demo.c        21105355 hello-name
21105369 myecho.c               21105380 ucontext_demo.c    21105371 waitpid_demo.c
21105370 execve_demo.c          21105372 fork_readfile      21105361 hello-name.c
21105378 pthread_yield_demo.c   21105377 vfork_demo
```

- 说明文件名没那么重要，一个文件可以有多个文件名
  - 只要inode编号一样，就是同一个文件

硬链接(hard link)

- 两个目录项
  - inode编号一样，
  - 名字不一样
- “text.txt”和“text2.txt”地位是相同的，内容也是一样的。
- 怎么做到的：

yxsu@Dell-T6401:~/os/process$ ln test.txt test2.txt



符号链接(symbolic link)

- 一个假的文件，只保存目标文件的路径

```
yxsu@Dell-T6401:~/os/process$ ln -s test.txt test3.txt
```

- 目标文件路径变了，这个链接也就失效了

- 有什么用：
  - 提供更加多元的文件读取方式

```
yxsu@Dell-T6401:/usr/lib/gcc/x86_64-linux-gnu/9$ file libstdc++.so
libstdc++.so: symbolic link to ../../../x86_64-linux-gnu/libstdc++.so.6

yxsu@Dell-T6401:/usr/lib/x86_64-linux-gnu$ ls libstdc++.so*
libstdc++.so.6  libstdc++.so.6.0.28
```

大纲

- 块设备
- 基于inode的文件系统
- 基于表的文件系统
  - FAT
  - NTFS
- 使用文件系统
  - 以命令行的形式使用
  - 在代码里使用
    - 基本操作
    - 内存映射
  - 文件系统的高级功能
- 虚拟文件系统
- 用户态文件系统

fork()的示例

```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

char str[11] = {0};

int main(int argc, char* argv[]) {
    int fd = open("test.txt", O_RDWR);
    if (fork() == 0) {
        ssize_t cnt = read(fd, str, 10);
        printf("Child process: %s\n", str);
    } else {
        ssize_t cnt = read(fd, str, 10);
        printf("Parent process: %s\n", str);
    }
    close(fd);
    return 0;
}
```

- test.txt文件中的内容:
  abcdefghijklmnopqrst

- 猜一猜程序运行的结果:

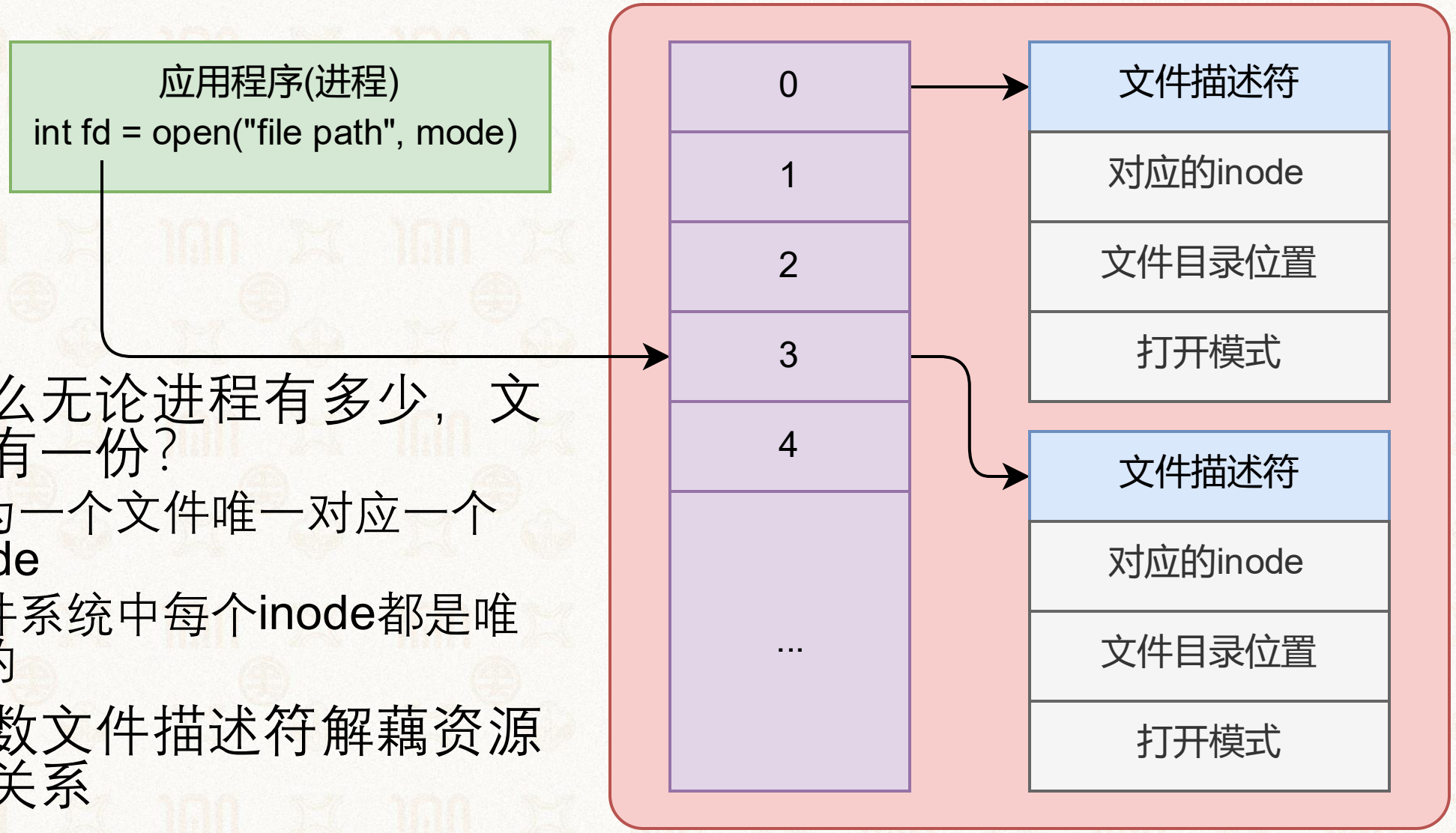
Child process: abcdefghijklmnopqrst
Parent process: abcdefghijklmnopqrst

Parent process: abcdefghijklmnopqrst
Child process: abcdefghijklmnopqrst

Child process: abcdefghij
Parent process: klmnopqrst

Parent process: abcdefghij
Child process: klmnopqrst

文件描述符



- 为什么无论进程有多少，文件只有一份？
  - 因为一个文件唯一对应一个inode
  - 文件系统中每个inode都是唯一的
- 用整数文件描述符解藕资源所属关系

使用文件接口读写文件

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

#define DATA_SIZE 20

int main() {
    int fd;
    char data[DATA_SIZE + 1];
    // 打开文件
    fd = open("/home/yxsu/os/demo.cpp", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    // 读取文件中的前20个字节
    read(fd, data, DATA_SIZE);
    // 打印文件中的前20个字符
    data[DATA_SIZE] = '\0';
    printf("file data: %s\n", data);
    // 向文件中写入6个字节的数据
    write(fd, "hello\n", 6);
    // 关闭文件
    close(fd);
    return 0;
}
```

文件读写模式　　文件的权限

复制文件

```c
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#define BUF_SIZE 4096
#define OUTPUT_MODE 0700
int main(int argc, char* argv[]) {
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];
    if(argc != 3) exit(1);
    in_fd = open(argv[1], O_RDONLY);
    if(in_fd < 0) exit(2);
    out_fd = creat(argv[2], OUTPUT_MODE);
    if(out_fd < 0) exit(3);

    while(1) { // copy
        rd_count = read(in_fd, buffer, BUF_SIZE);
        if(rd_count <0) break;
        wt_count = write(out_fd, buffer, rd_count);
        if(wt_count <= 0) exit(4);
    }
    close(in_fd); close(out_fd);
    exit(0);
}
```

- ./copyfile copyfile.c copyfile2.c
  - 将copyfile.c复制成copyfile2.c

文件操作的相关接口


```
// 打开文件
int open(const char* path, int oflag, ...);
int openat(int fd, const char *path, int offlag, ...);

// 读写文件
ssize_t read(int fildes, void* buf, size);
ssize_t write(int fildes, const void* buf, size_t nbyte);
// 调整读写位置
off_t lseek(int fildes, off_t offset, int whence);

// 获取文件属性
int fstat(int fildes, struct stat *buf);
int fstatat(int fd, const char* restrict path, struct stat* restrict buf, int flag);
int lstat(const char* restrict path, struct stat* restrict buf);
int stat(const char* restrict path, struct stat* );

// 关闭文件
int close(int flides);
```

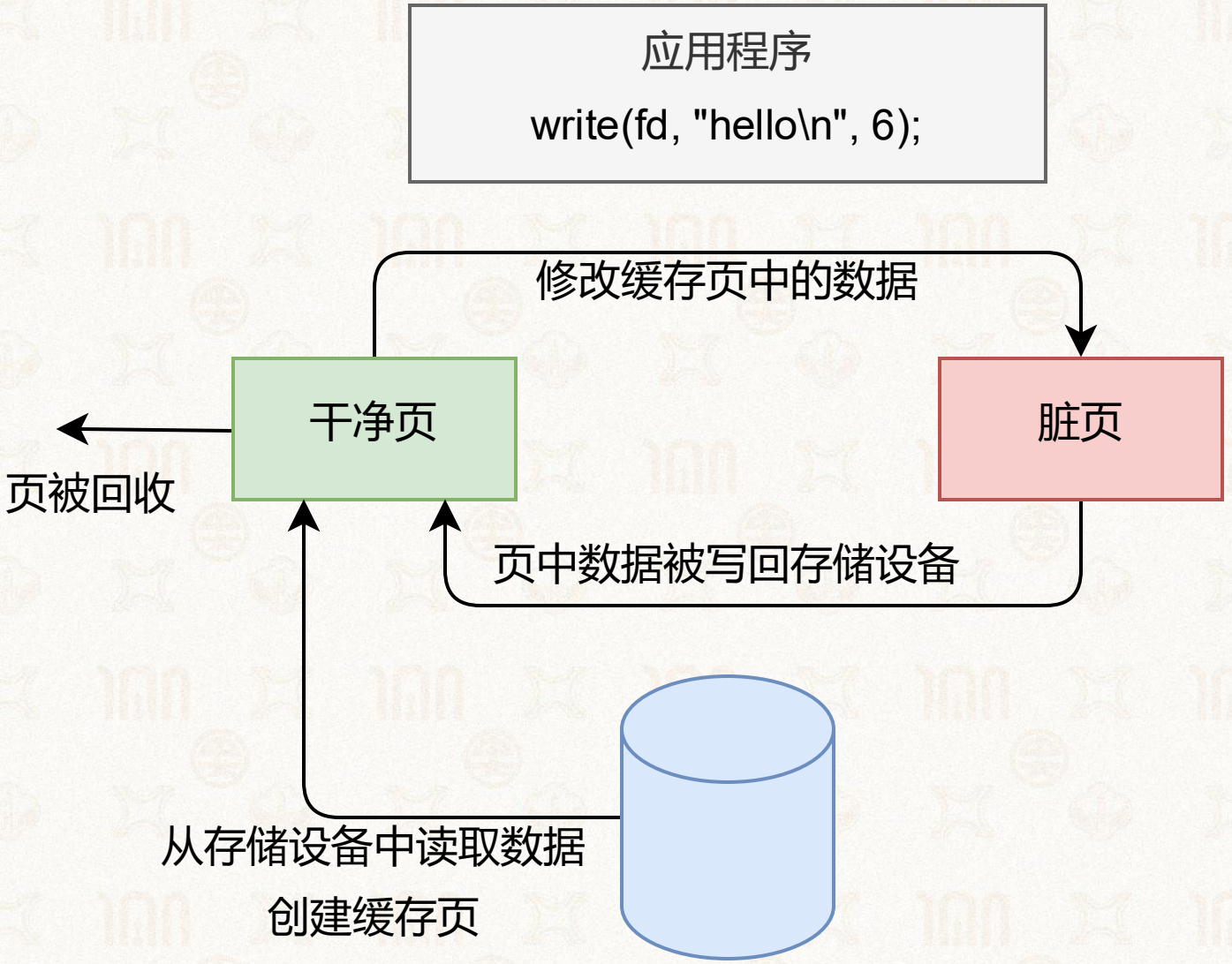
页缓存与脏页

- 为提高读写效率，操作系统会设置文件缓存
  - 不用每次read/write都去读写硬盘
  - 数据没有真的写在硬盘中，而是写在内存中缓存

- 怕突然断电、怕电脑死机，不想让内存缓存怎么办？

```
int fsync(int fildes);
```

- 强制将缓存内容写入硬盘中



大纲

- 块设备
- 基于inode的文件系统
- 基于表的文件系统
  - FAT
  - NTFS
- 使用文件系统
  - 以命令行的形式使用
  - 在代码里使用
    - 基本操作
    - 内存映射
  - 文件系统的高级功能
- 虚拟文件系统
- 用户态文件系统

内存映射接口

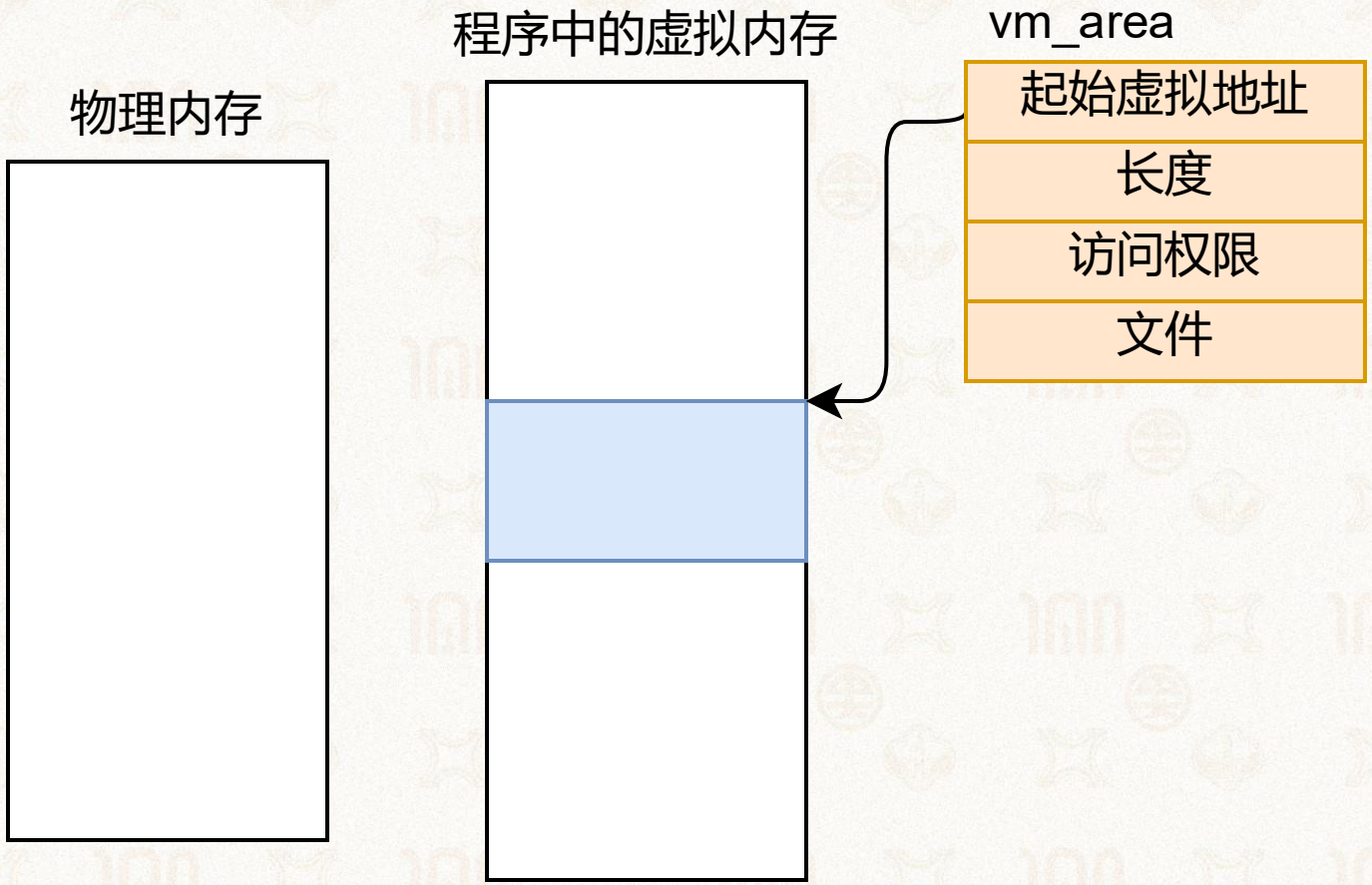
- 既然内存可以当作缓存，不如把文件一次性全部加载入内存中
- mmap可将文件映射到虚拟内存空间中
- 常用接口：

```
fd = open("/home/yxsu/os_file", O_RDWR);
addr = mmap(NULL, length, PROT_WRITE, MAP_SHARED, fd, 0);
memset(addr, 0, length);
```

```
void* nmap(void* addr, size_t len, int prot, int flags, int fildes, off_t off);
int msync(void* addr, size_t len, int flags);
int munmap(void* addr, size_t len);
```

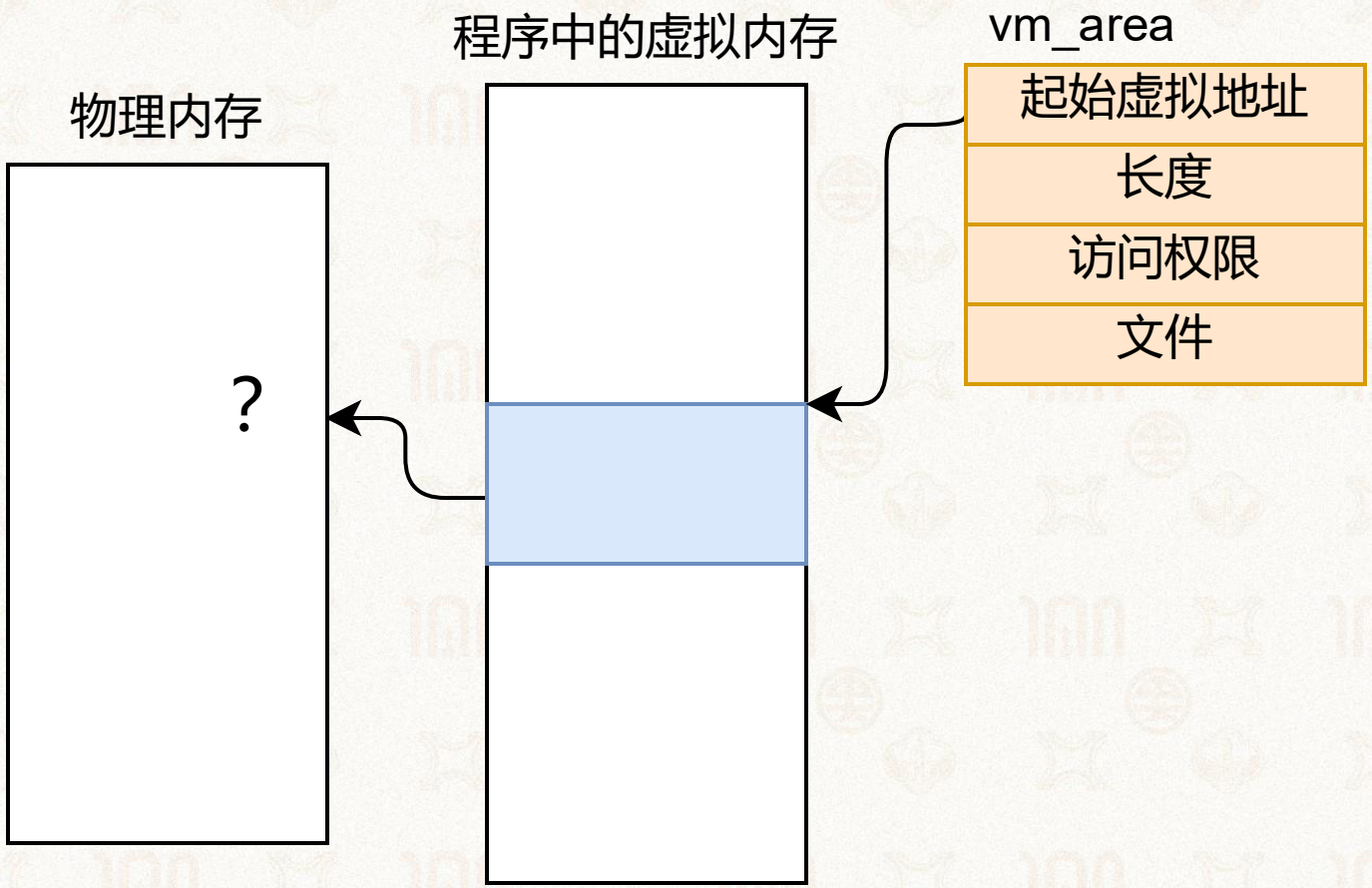
实现内存映射

- 调用mmap时分配虚拟地址
- 标记此段虚拟地址与该文件的inode编号



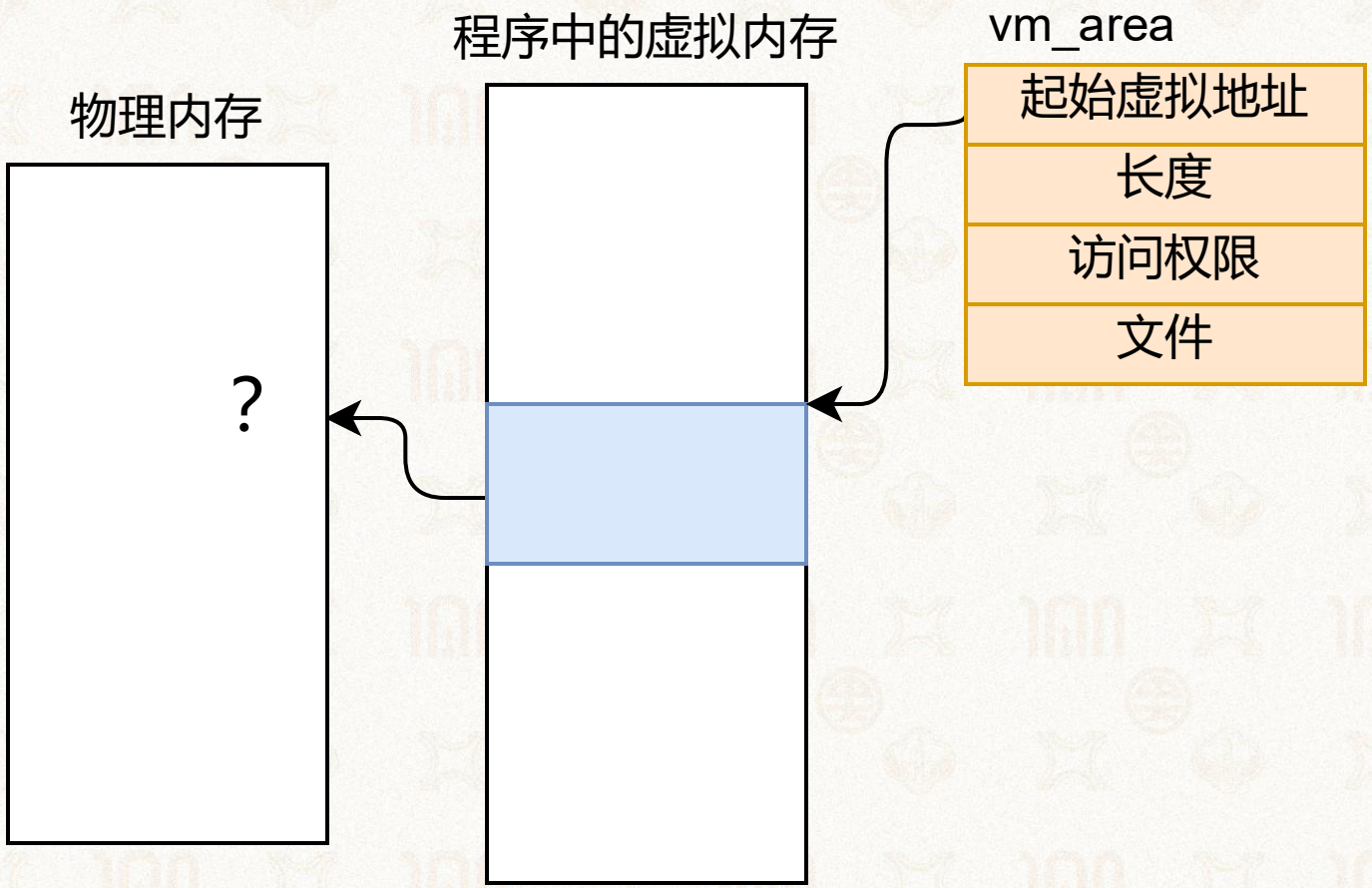
实现内存映射

- 访问mmap返回的虚拟地址时，触发缺页中断（page fault）



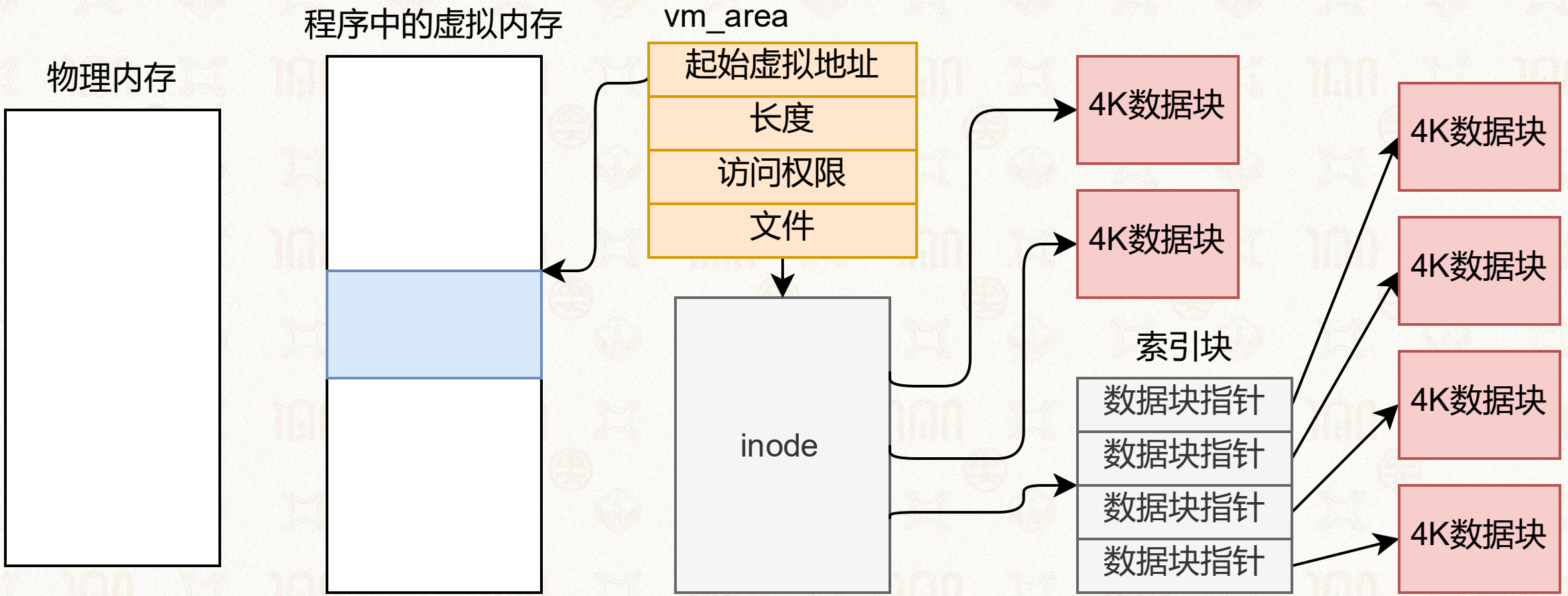
实现内存映射

- 缺页中断处理函数，通过虚拟地址，找到该文件的inode



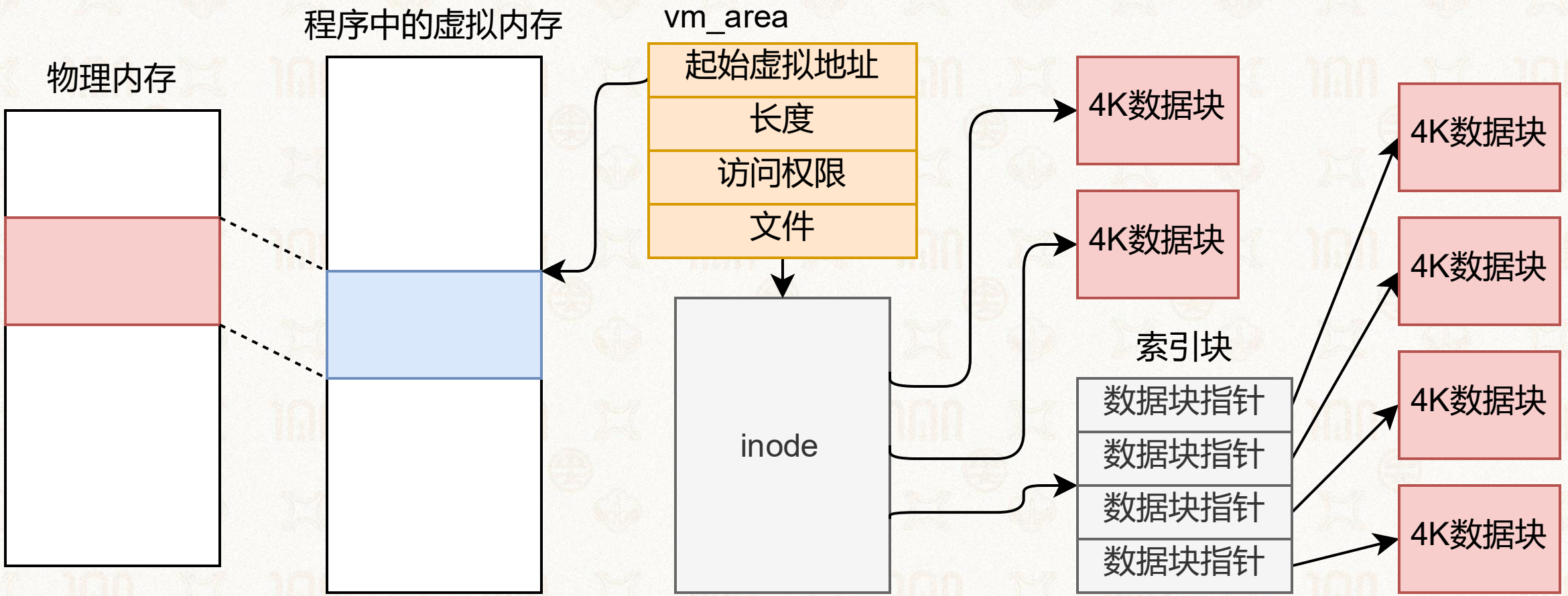
实现内存映射

- 缺页中断处理函数，通过虚拟地址，找到该文件的inode



实现内存映射

- 从磁盘中将inode中对应的数据读到内存页中



文件内存映射的优势

- 对于随机访问，不用频繁lseek
- 减少系统调用次数
- 可以减少数据copy
  - 如拷贝文件，数据无需经过中间buffer
- 访问的局部性更好
- 可以用madvice为内核提供访问提示，提高性能

大纲

- 块设备
- 基于inode的文件系统
- 基于表的文件系统
  - FAT
  - NTFS
- 使用文件系统
  - 以命令行的形式使用
  - 在代码里使用
    - 基本操作
    - 内存映射
  - 文件系统的高级功能
- 虚拟文件系统
- 用户态文件系统

文件复制

- 正常的文件复制：read / write
- 基于内存映射的文件复制

yxsu@Dell-T6401:~/os$ cp A B

- 1. 打开文件A
- 2. 创建并打开文件B
- 3. 从A中读出数据到buffer
- 4. 将buffer中的数据写入B
- 5. 重复3、4直到文件A被读完

- 1. 打开文件A
- 2. 获取A的大小为X
- 3. 创建并打开文件B
- 4. 改变B的大小为X
- 5. 将A和B分别mmap到内存空间
- 6. memcpy

基于内存映射的文件复制

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
int main(int argc, char *argv[]) {
    int input, output; size_t filesize;
    void *source, *target;
    input = open(argv[1], O_RDONLY);
    output = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, 0666)

    filesize = lseek(input, 0, SEEK_END);
    lseek(output, filesize - 1, SEEK_SET);
    write(output, '\0', 1);

    if ((source = mmap(0, filesize, PROT_READ, MAP_SHARED, input, 0)) == (void *)-1)
        fprintf(stderr, "Error mapping input file: %s\n", argv[1]), exit(1);
    if ((target = mmap(0, filesize, PROT_WRITE, MAP_SHARED, output, 0)) == (void *)-1)
        fprintf(stderr, "Error mapping ouput file: %s\n", argv[2]), exit(1);

    memcpy(target, source, filesize);

    munmap(source, filesize); munmap(target, filesize);
    close(input); close(output); return 0;
}
```

(void *)-1) 表示0xFFFF……FFFF

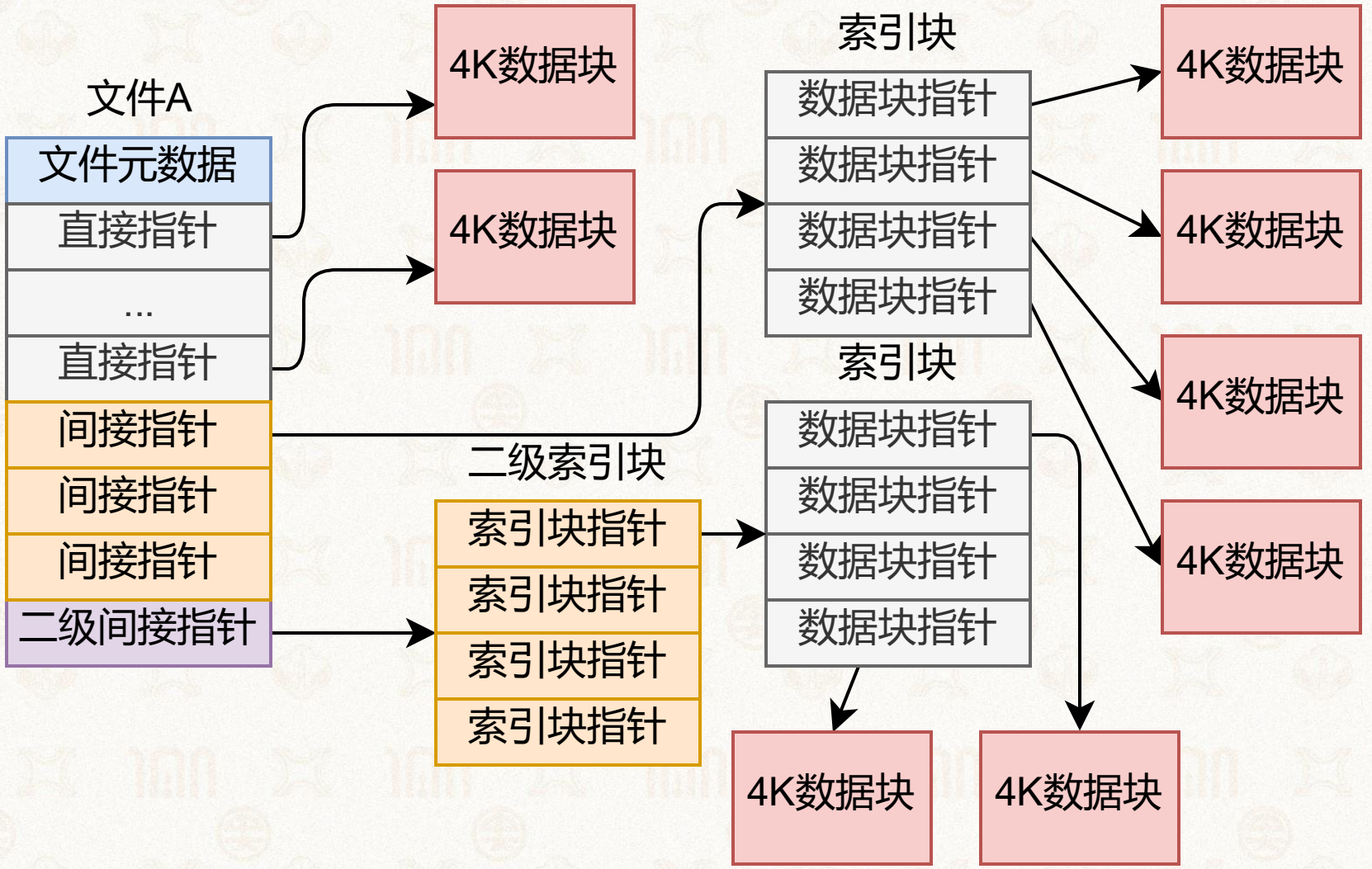
使用lseek定位操作，在最后一位写入'\0'

内存映射核心操作

复制内存空间数据

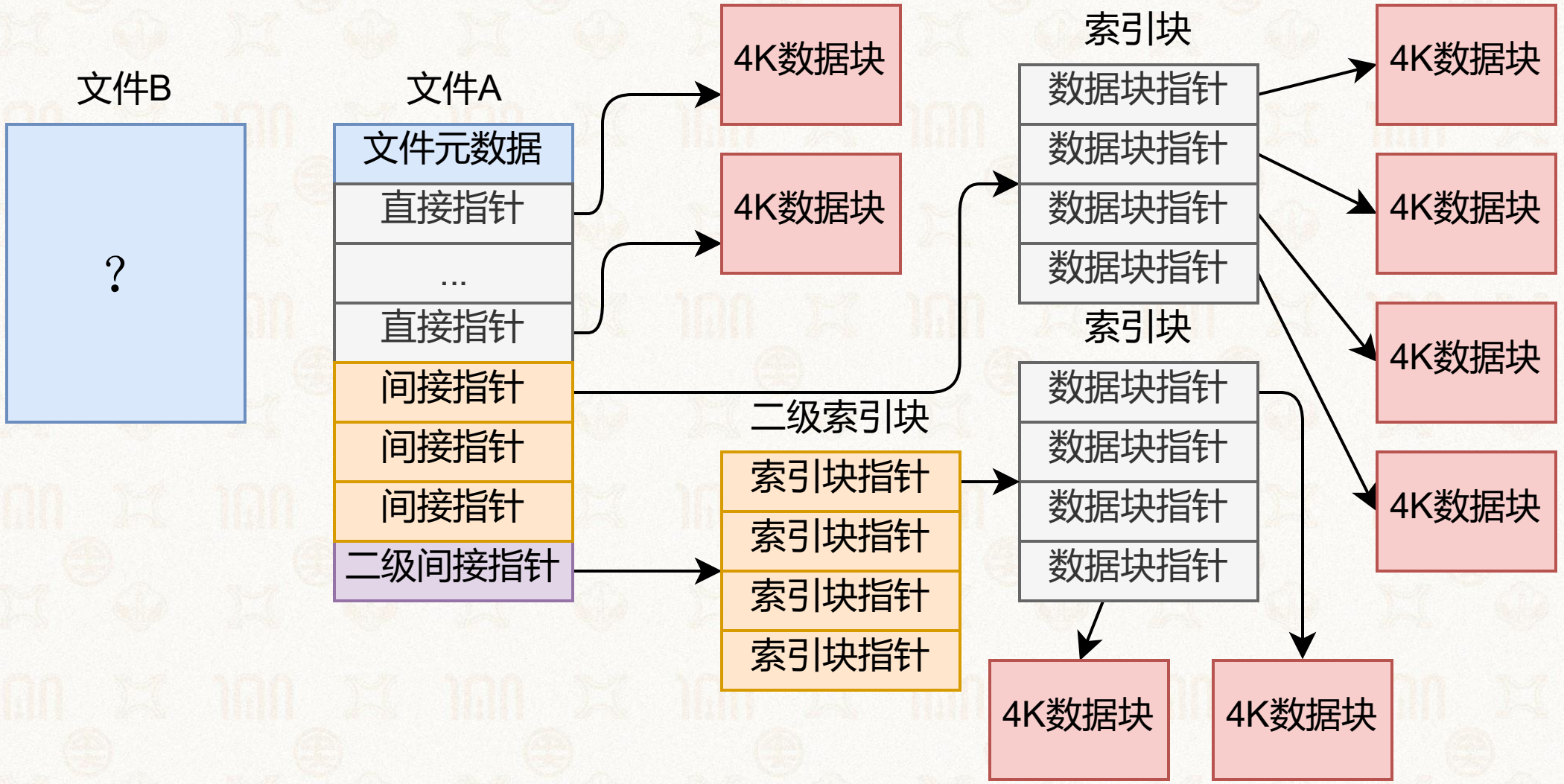
文件克隆

- 学完这个复杂结构，总要发挥点作用



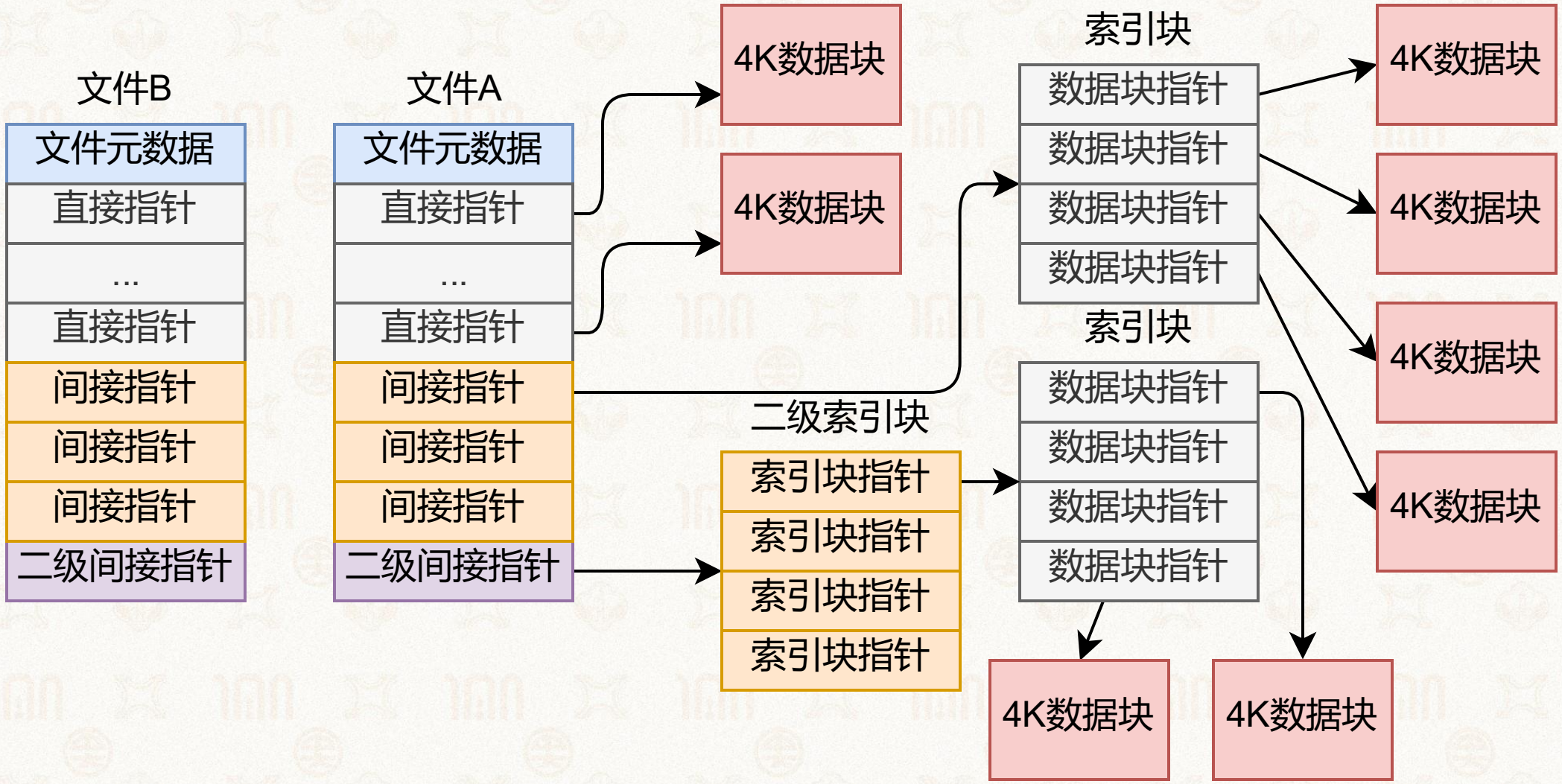
文件克隆

- 如何实现基于文件系统的快速复制?



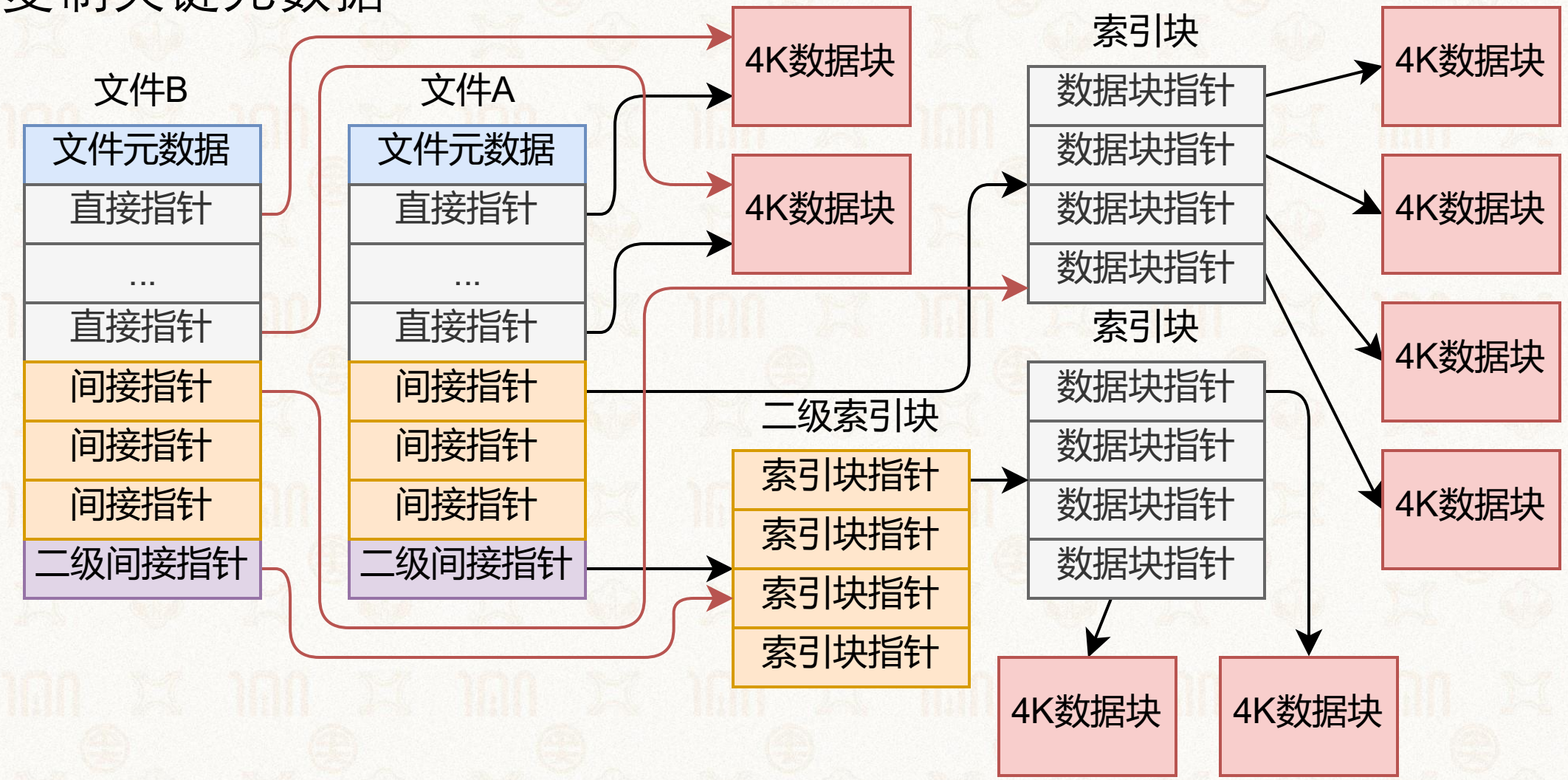
文件克隆

- 文件系统层面的复制



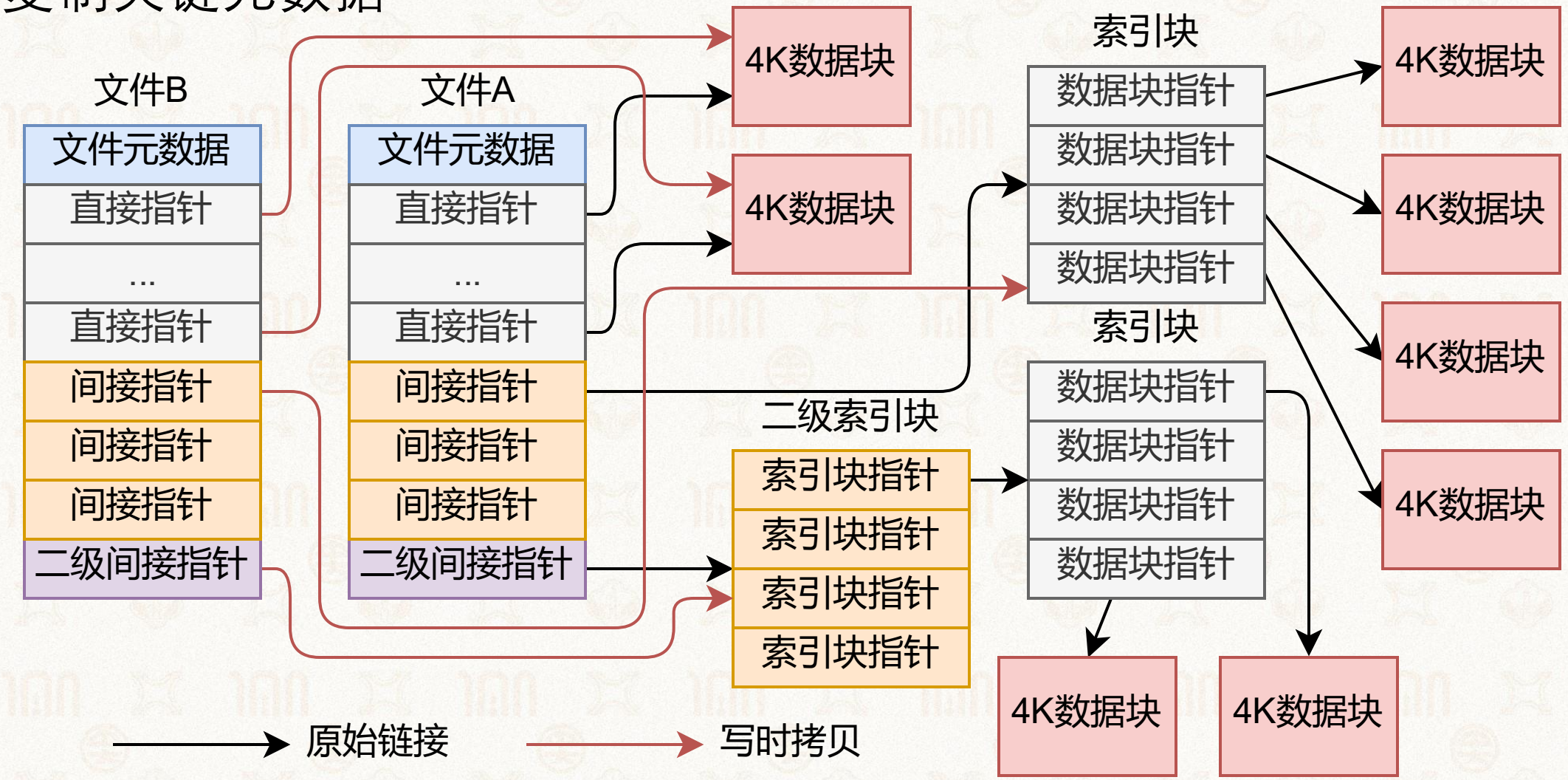
文件克隆

- 文件系统层面的复制
- 只复制关键元数据



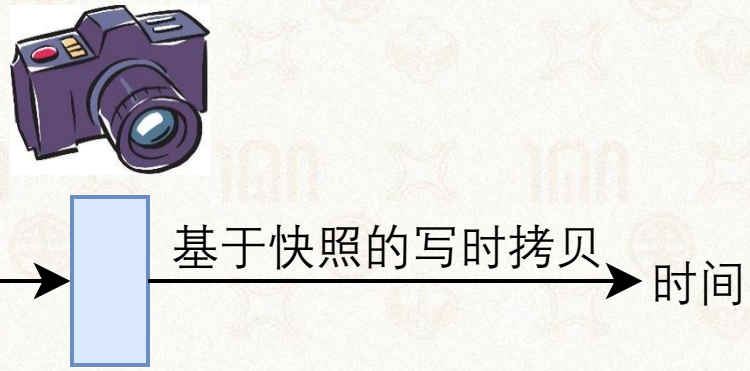
文件克隆

- 文件系统层面的复制
- 只复制关键元数据
- 其它部分“写时拷贝”共享



快照(snapshot)

- 同样使用“写时拷贝”
- 对于基于inode表的文件系统
  - 将inode表拷贝一份作为快照保存
  - 标记已用数据区为“写时拷贝”
- 对于树状结构的文件系统
  - 将树根拷贝一份作为快照保存
  - 树根以下的节点标记为“写时拷贝”



文件系统的一些其他高级功能

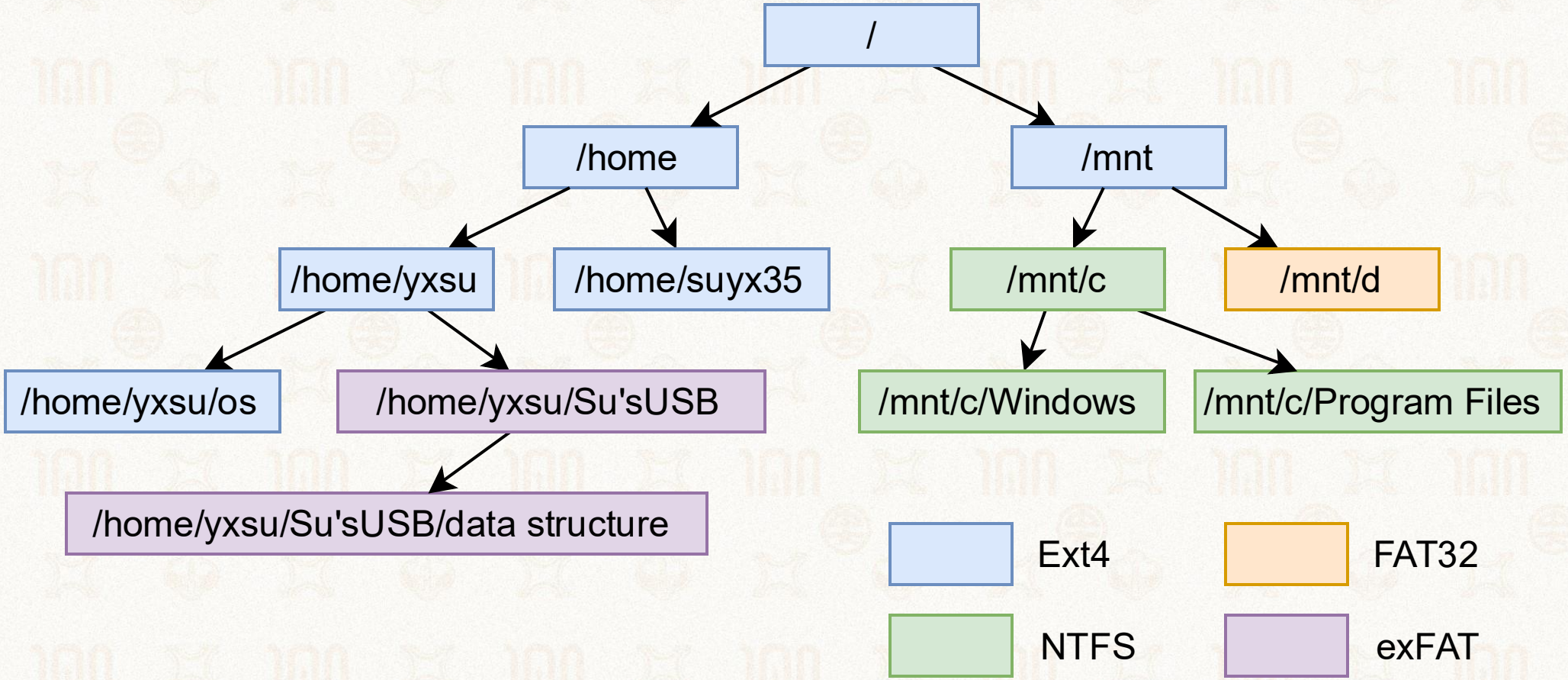
- 加密
- 压缩
- 去重
- 数据和元数据校验
- 配额管理（QoS）
- 软件RAID
- 多设备管理
- 子卷
- 事务（Transaction）

大纲

- 块设备
- 基于inode的文件系统
- 基于表的文件系统
  - FAT
  - NTFS
- 使用文件系统
  - 以命令行的形式使用
  - 在代码里使用
    - 基本操作
    - 内存映射
  - 文件系统的高级功能
- 虚拟文件系统
- 用户态文件系统

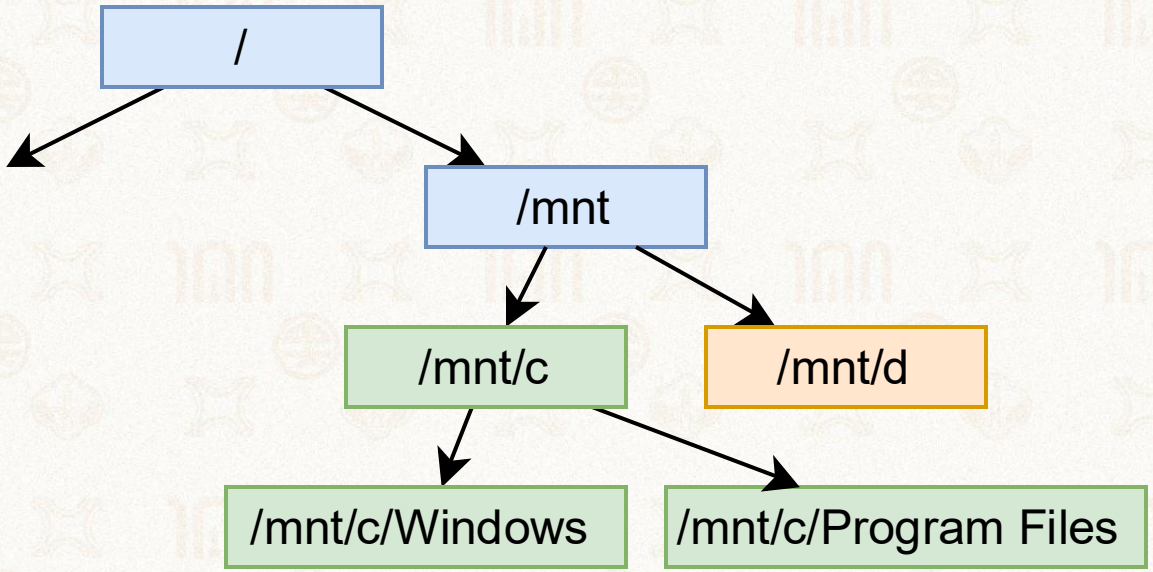
多种文件系统的挂载

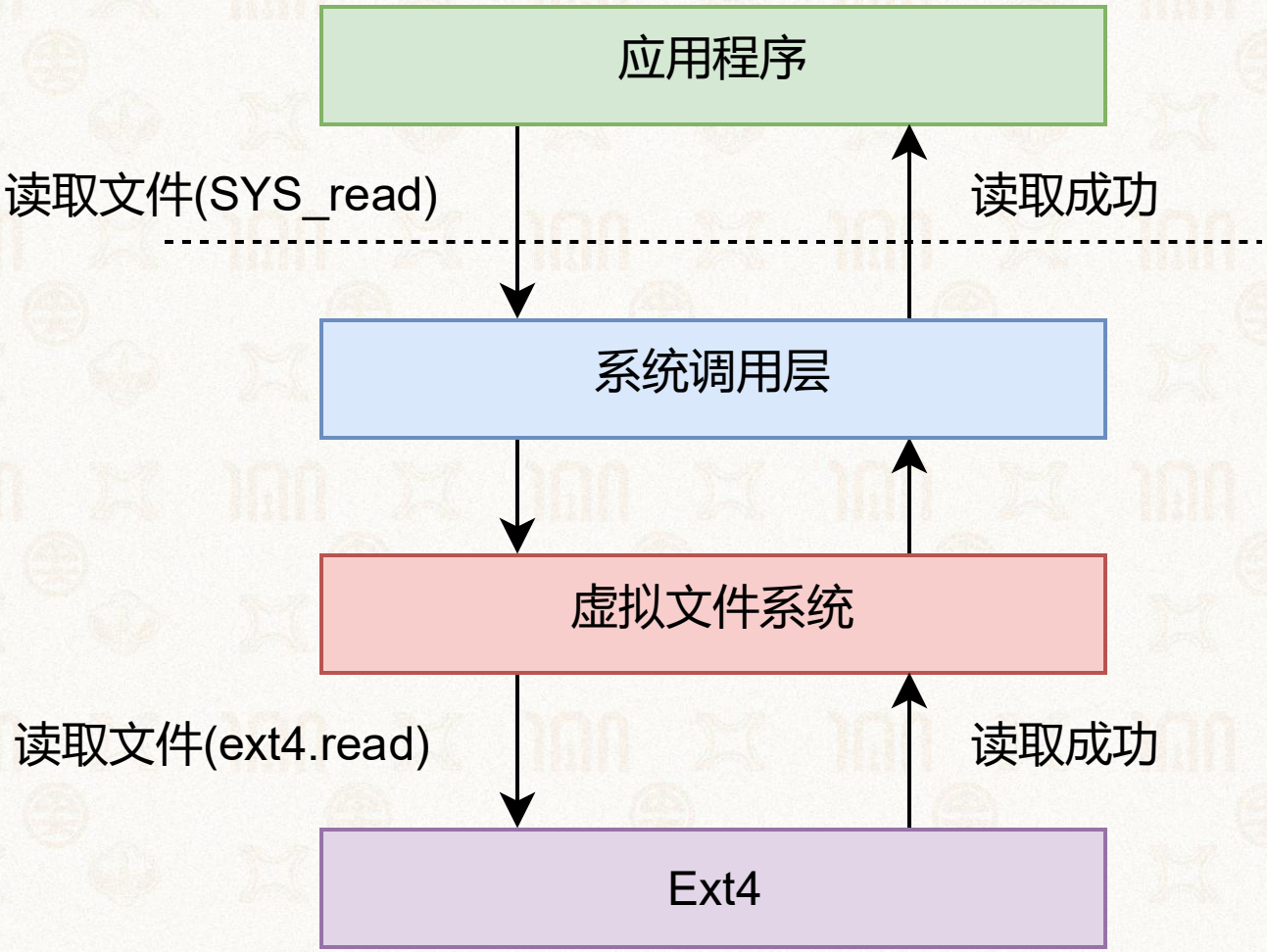
- 每种文件系统都有：
  - 超级块(元数据)
  - 根目录
  - 树状结构



Linux VFS(Virtual File System)：虚拟文件系统

- Linux系统的VFS可以统一挂载
  - Linux的VFS定义了一些系列接口
  - 具体的文件系统实现这些接口
- 如在读取一个inode的文件时
  - VFS先找到该inode所属文件系统
  - 再调用该文件系统的读取接口





Linux VFS中的挂载结构

```
struct vfsmount {
    struct dentry *mnt_root;     // 挂载树的根目录
    struct super_block *mnt_sb; // 指向超级块的指针
    int mnt_flags;
    struct user_namespace *mnt_userns;
} __randomize_layout;
```

- 挂载的文件系统

```
struct mountpoint {
    struct hlist_node m_hash;
    struct dentry *m_dentry;  // 挂载点所在的目录项
    struct hlist_head m_list;
    int m_count;
};
```

- 具体某个挂载点

https://elixir.bootlin.com/linux/v5.16.14/source/include/linux/mount.h#L71
https://elixir.bootlin.com/linux/v5.16.14/source/fs/mount.h#L32

Linux VFS中的挂载结构

```
struct mount {
    struct hlist_node mnt_hash;
    struct mount *mnt_parent;
    struct dentry *mnt_mountpoint;
    struct vfsmount mnt;
    // ...
    struct list_head mnt_mounts;    /* list of children, anchored here */
    struct list_head mnt_child; /* and going through their mnt_child */
    struct list_head mnt_instance;  /* mount instance on sb->s_mounts */
    const 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 mount *mnt_master;   /* slave is on master->mnt_slave_list */
    struct mnt_namespace *mnt_ns;   /* containing namespace */
    struct mountpoint *mnt_mp;  /* where is it mounted */
    union {
        struct hlist_node mnt_mp_list;  /* list mounts with the same mountpoint */
        struct hlist_node mnt_umount;
    };
    // ...
} __randomize_layout;
```

文件系统

挂载点的目录项

相同挂载点上的其它文件系统

https://elixir.bootlin.com/linux/v5.16.14/source/fs/mount.h#L39

Linux VFS 虚拟文件系统接口

- Linux的VFS定义的一些inode上的操作接口

```
struct inode_operations {
    // ...
    int (*create) (struct user_namespace *, struct inode *,struct dentry *, umode_t, bool);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct user_namespace *, struct inode *,struct dentry *, const char *);
    int (*mkdir) (struct user_namespace *, struct inode *,struct dentry *, umode_t);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*rename) (struct user_namespace *, struct inode *, struct dentry *, struct inode *, struct
dentry *, unsigned int);
    int (*setattr) (struct user_namespace *, struct dentry *, struct iattr *);
    int (*getattr) (struct user_namespace *, const struct path *, struct kstat *, u32, unsigned int);
    // ...
} ____cacheline_aligned;
```

https://elixir.bootlin.com/linux/v5.16.14/source/include/linux/fs.h#L2112

Linux VFS 虚拟文件系统接口

- Ext4文件系统对这些接口的实现

```
const struct inode_operations ext4_dir_inode_operations = {
    .create     = ext4_create,
    .lookup     = ext4_lookup,
    .link       = ext4_link,
    .unlink     = ext4_unlink,
    .symlink    = ext4_symlink,
    .mkdir      = ext4_mkdir,
    .rmdir      = ext4_rmdir,
    .mknod      = ext4_mknod,
    .tmpfile    = ext4_tmpfile,
    .rename     = ext4_rename2,
    .setattr    = ext4_setattr,
    .getattr    = ext4_getattr,
    .listxattr  = ext4_listxattr,
    .get_acl    = ext4_get_acl,
    .set_acl    = ext4_set_acl,
    .fiemap         = ext4_fiemap,
    .fileattr_get   = ext4_fileattr_get,
    .fileattr_set   = ext4_fileattr_set,
};
```

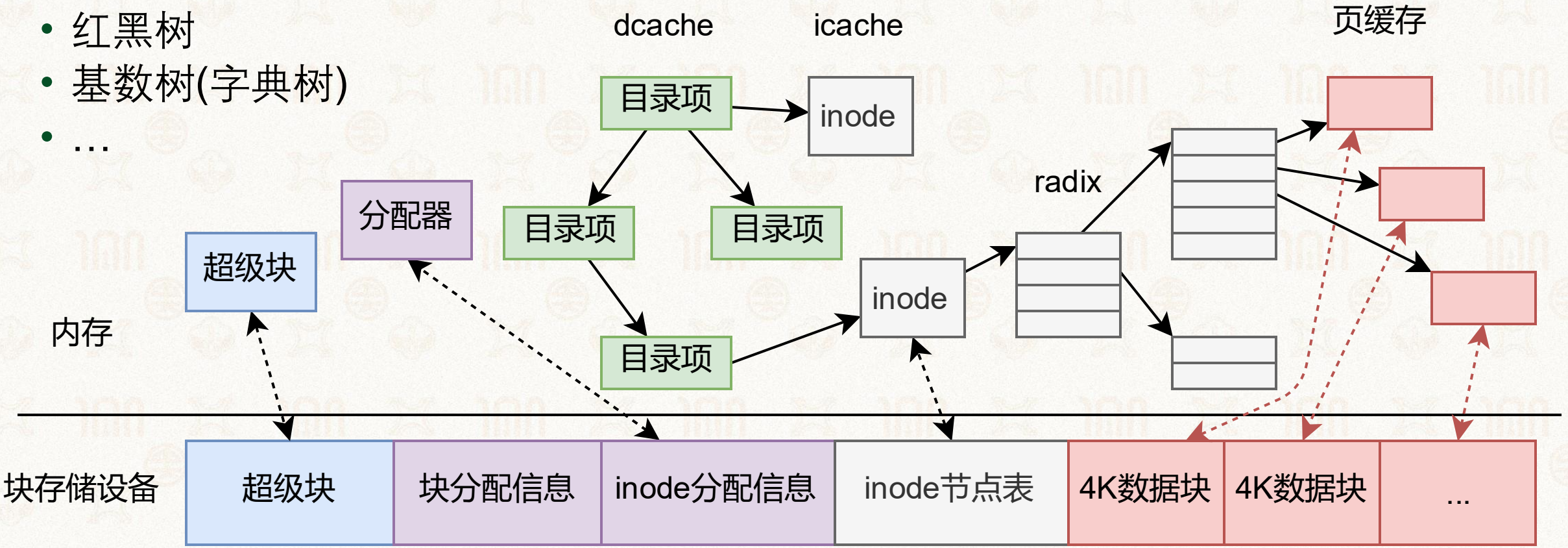
create的实现：
https://elixir.bootlin.com/linux/v5.16.14/source/fs/ext4/namei.c#L2732

mkdir的实现：
https://elixir.bootlin.com/linux/v5.16.14/source/fs/ext4/namei.c#L2912

https://elixir.bootlin.com/linux/v5.16.14/source/fs/ext4/namei.c#L4106

文件系统在内存中

- 文件系统中多种结构在内存中均有映射
- 利用更多数据结构做优化
  - 红黑树
  - 基数树(字典树)
  - …

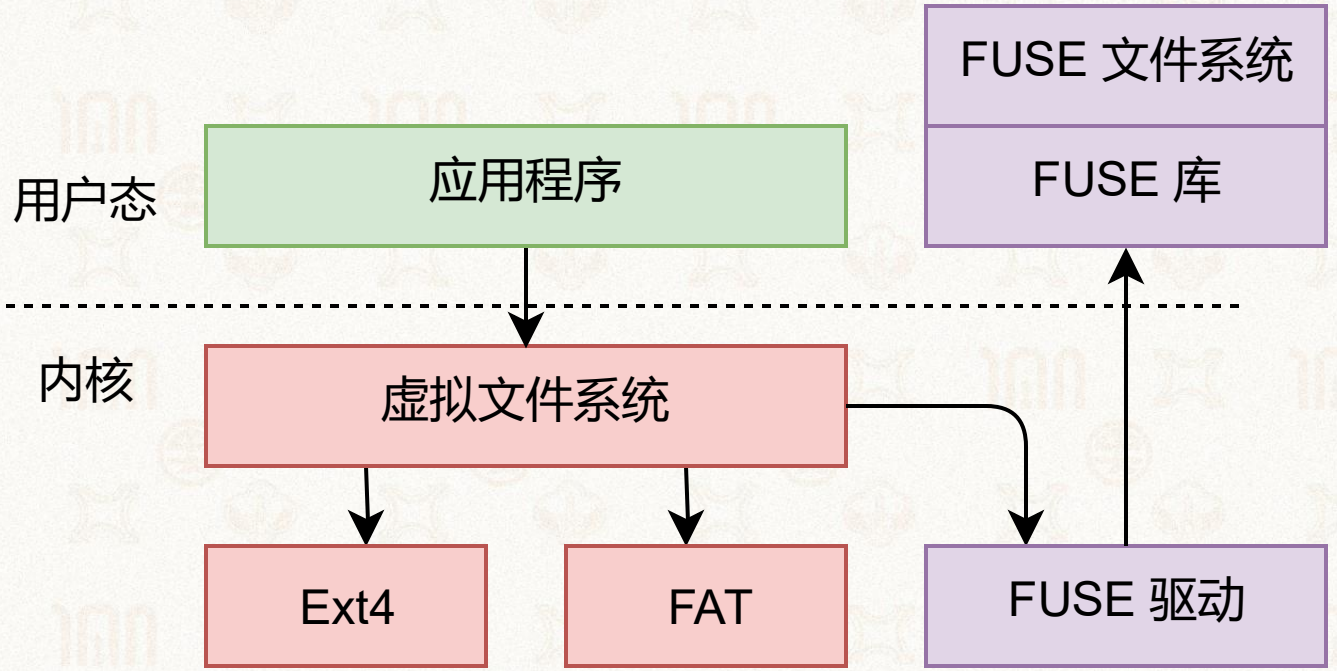


大纲

- 块设备
- 基于inode的文件系统
- 基于表的文件系统
  - FAT
  - NTFS
- 使用文件系统
  - 以命令行的形式使用
  - 在代码里使用
    - 基本操作
    - 内存映射
  - 文件系统的高级功能
- 虚拟文件系统
- 用户态文件系统

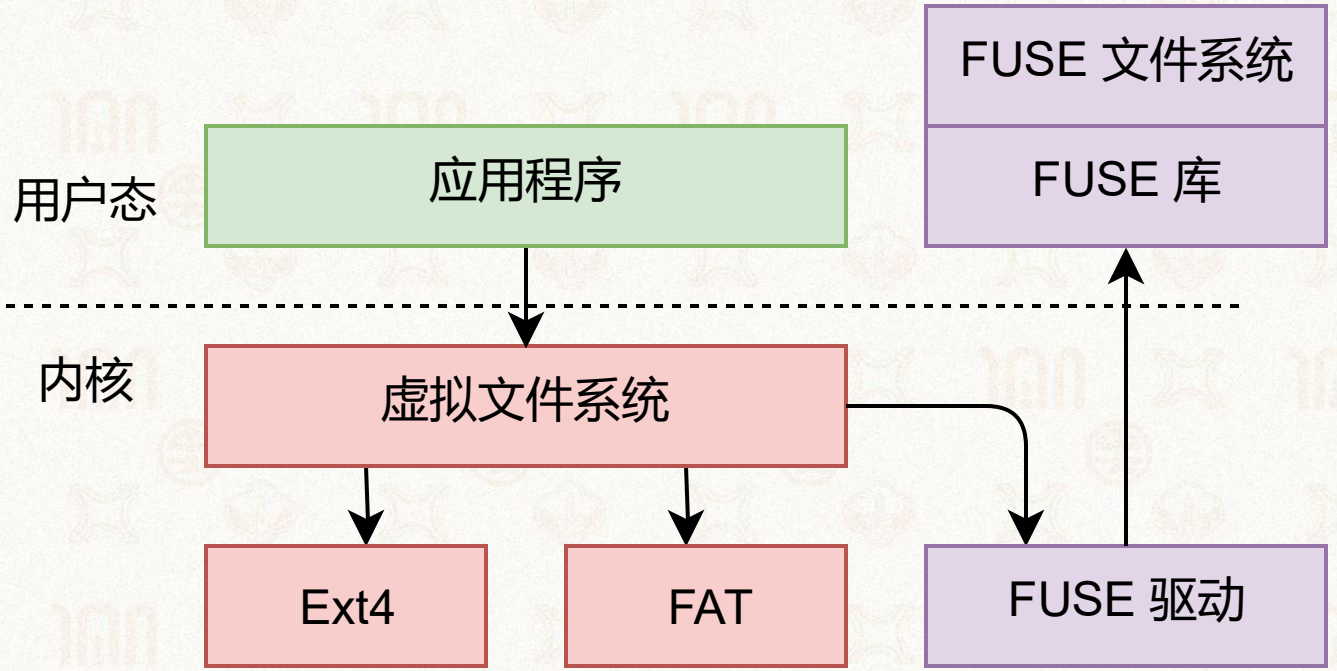
FUSE用户态文件系统框架

- 为什么要用户态文件系统?
  - 快速试验文件系统新设计
  - 大量第三方库可以使用
  - 方便调试
  - 无需担心把内核搞崩溃
  - 实现新功能



FUSE用户态文件系统框架

- 基本流程
  - FUSE文件系统向FUSE驱动注册（挂载）
  - 应用程序发起文件请求
  - 根据挂载点，VFS将请求转发给FUSE驱动
  - FUSE驱动通过中断、共享内存等方式将请
  - 求发给FUSE文件系统
  - FUSE文件系统处理请求
  - FUSE文件系统通知FUSE驱动请求结果
  - FUSE驱动通过VFS返回结果给应用程序



从这个流程中可以看出FUSE有什么问题?

FUSE 底层API

```
int main(int argc, char *argv[]) {
    struct fuse_args args = FUSE_ARGS_INIT(argc, argv);
    struct fuse_session *se;
    // ...
    se = fuse_session_new(&args, &hello_ll_oper,
                          sizeof(hello_ll_oper), NULL);
    // ...
    if (fuse_session_mount(se, opts.mountpoint) != 0)
        goto err_out3;

    /* Block until ctrl+c or fusermount -u */
    if (opts.singlethread) {
        ret = fuse_session_loop(se);
    }
    else {
        config.clone_fd = opts.clone_fd;
        config.max_idle_threads = opts.max_idle_threads;
        ret = fuse_session_loop_mt(se, &config);
    }
    // ...
    return ret ? 1 : 0;
}
```

建立连接

挂载

等待请求

像一个普通程序一样写文件系统

完整代码见: http://libfuse.github.io/doxygen/example_2hello__ll_8c.html

FUSE 高层API

```c
static int hello_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi) {
    size_t len;
    (void)fi;
    if (strcmp(path + 1, options.filename) != 0)
        return -ENOENT;
    len = strlen(options.contents);
    if (offset < len) {
        if (offset + size > len)
            size = len - offset;
        memcpy(buf, options.contents + offset, size); // 只是在buf中填数据
    } else {
        size = 0;
    }
    return size;
}

int main(int argc, char *argv[])
{
    // ...
    ret = fuse_main(args.argc, args.argv, &hello_oper, NULL); // 挂载、注册并等待请求
    // ...
    return ret;
}
```

完整代码见: http://libfuse.github.io/doxygen/example_2hello_8c.html

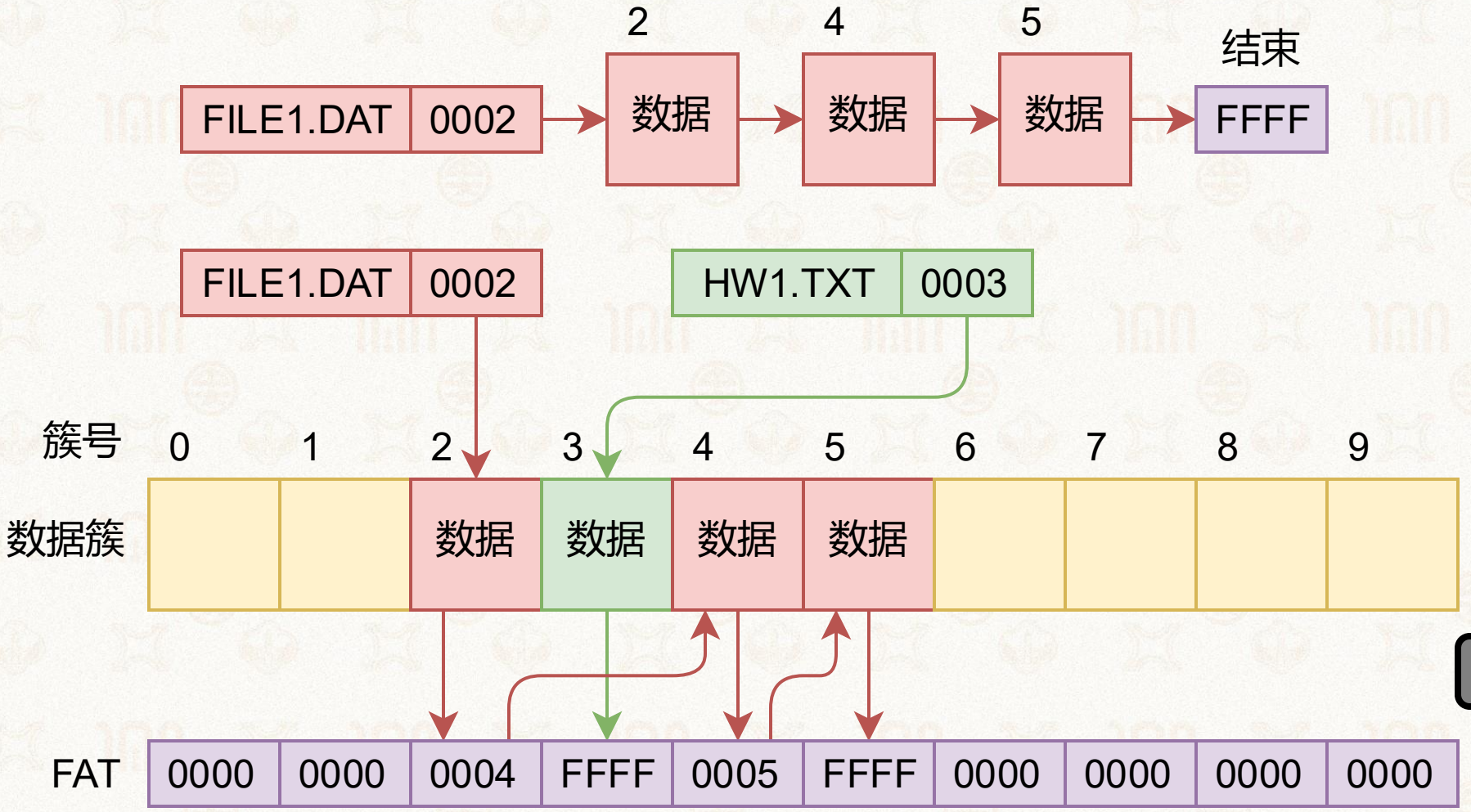
现实中，FUSE能用来做什么？

- SSHFS（用ssh挂载远端目录到本地）
- NTFS-3G
- GMailFs（以文件接口收发邮件）
- WikipediaFS（用文件查看和编辑Wikipedia）
- 网盘同步
- 分布式文件系统（Lustre、GlusterFS等）

主观题 10分

设置

FAT 文件系统中并不支持文件硬链接。在 FAT 文件系统实现文件硬链接有哪些困难？如何修改能解决这些困难？



作答

谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course
匿名提问箱：https://suask.me/ask-teacher/106/苏玉鑫