IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学 SHANGHAI JIAO TONG UNIVERSITY

文件系统API实现与崩溃一致性

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

文件系统API与元数据

- **应用程序通过系统调用使用这些 API**
  - CHDIR, MKDIR, RMDIR
  - CREAT, LINK, UNLINK, RENAME
  - SYMLINK
  - MOUNT, UNMOUNT
  - OPEN, READ, WRITE, CLOSE
  - SYNC
- **文件系统的两类元数据**
  - 磁盘上文件的元数据：静态的、在磁盘中
  - 被打开文件的元数据：动态的、在内存中

文件的元数据 （磁盘中）

- **拥有者/所在组 ID**
  - 拥有该 inode 的 用户 ID 和 组 ID
- **权限的类型**
  - 拥有者、所在组、其他
  - 读、写、执行
- **时间戳**
  - 最后一次访问 (如READ 操作)
  - 最后一次修改 (如WRITE 操作)
  - 最后一次 inode 更新 (如LINK 操作)

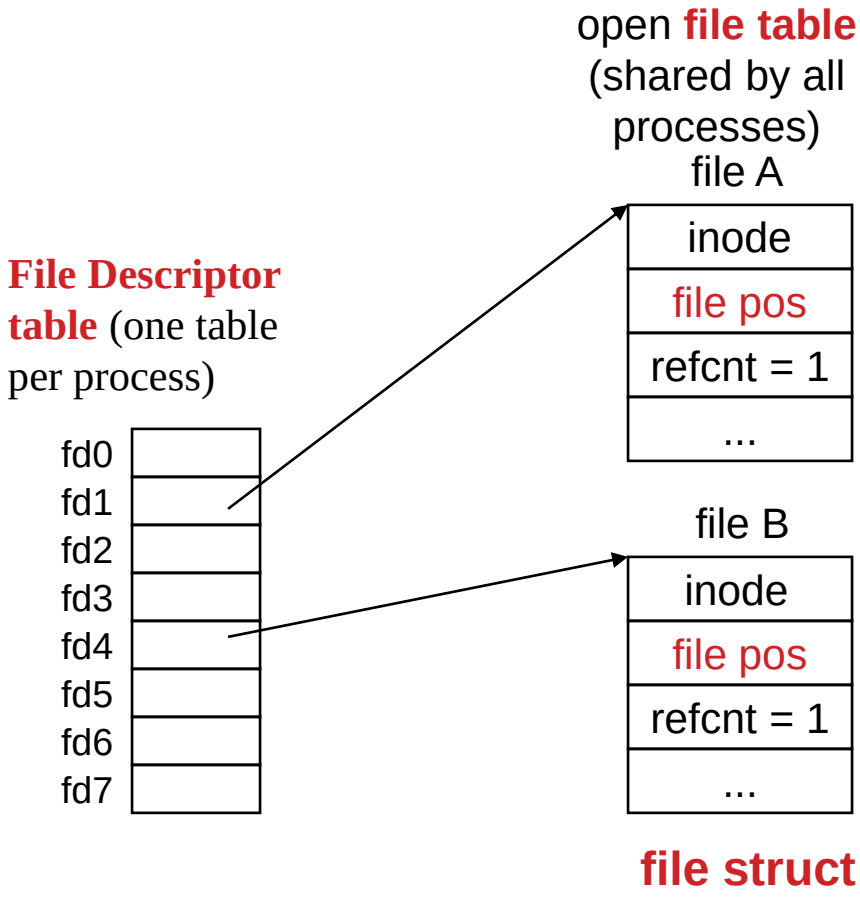
POSIX定义的部分文件元数据（inode）

文件元数据	说　　明
mode	文件模式，其中包括文件类型和文件权限
nlink	指向此文件的链接个数
uid	文件所属用户的 ID
gid	文件所属用户组的 ID
size	文件的大小
atime	文件数据最近访问时间
ctime	文件元数据最近修改时间
mtime	文件数据最近修改时间

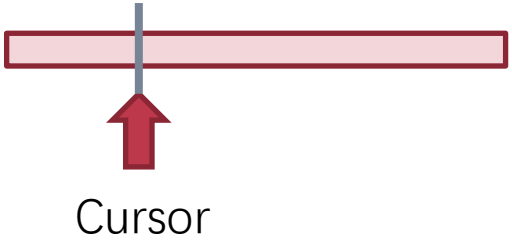
被打开文件的元数据（内存中）

- **整个系统维护了一个 file_table**
  - 记录了所有打开的文件的信息
  - 包括：文件游标（file cursor）、引用数（ref_count）
  - 父子进程间可以共享文件游标
- **每个进程维护了一个 fd_table**
  - 记录了该进程每个 **fd** 所对应文件在 **file_table** 中的索引

fd table 和 file table



文件游标 Cursor



- **文件游标**
  - 记录了一个文件中下一次操作的位置
  - 可以通过 **SEEK** 操作修改
- **情况 1: 共享游标**
  - 父进程将 fd 传递给子进程
    - UNIX 中，子进程会继承父进程所有已经打开的 fd
  - 允许父子进程共享同一个文件
- **情况 2: 非共享游标**
  - 两个不同的进程打开同一个文件

lseek示例

```c
int main()
{
  int fd = open("file.txt", O_RDWR | O_CREAT, 0664);

  const char* data = "hello, world\n";
  write(fd, data, strlen(data));

  // 使用lseek函数移动文件的读写指针
  lseek(fd, 0, SEEK_SET);

  // 使用read函数从文件中读取数据
  char buffer[1024] = {0};
  read(fd, buffer, sizeof(buffer));

  printf("%s", buffer);

  close(fd);
  return 0;
}
```

运行结果：

```
➜  os-fs ls
a.out  test.c
➜  os-fs ./a.out
hello, world
➜  os-fs ls
a.out    file.txt test.c
➜  os-fs cat file.txt
hello, world
```

lseek示例

```c
int main()
{
  int fd = open("file.txt", O_RDWR | O_CREAT, 0664);

  const char* data = "hello, world\n";
  write(fd, data, strlen(data));

  // 使用lseek函数移动文件的读写指针
  //lseek(fd, 0, SEEK_SET);

  // 使用read函数从文件中读取数据
  char buffer[1024] = {0};
  read(fd, buffer, sizeof(buffer));

  printf("%s", buffer);

  close(fd);
  return 0;
}
```

运行结果：

```
→  os-fs ls
a.out   a2.out  test.c  test2.c
→  os-fs ./a2.out
→  os-fs ls
a.out    a2.out   file.txt test.c   test2.c
→  os-fs cat file.txt
hello, world
```

练习：lseek

```c
int main()
{
  int fd = open("file.txt", O_RDWR | O_CREAT, 0664);

  const char* data = "hello, world\n";
  write(fd, data, strlen(data));

  // 使用lseek函数移动文件的读写指针
  lseek(fd, 0, SEEK_SET);

  // 使用read函数从文件中读取数据
  char buffer[1024] = {0};
  read(fd, buffer, sizeof(buffer));

  printf("%s", buffer);

  // 使用lseek函数移动文件的读写指针
  lseek(fd, 0, SEEK_SET);
  const char* data1 = "zsxg";
  write(fd, data1, strlen(data1));

  // 使用lseek函数移动文件的读写指针
  lseek(fd, 0, SEEK_SET);
  read(fd, buffer, sizeof(buffer));

  printf("%s", buffer);

  close(fd);
  return 0;
}
```

思考：注释第28行，会读到什么？

运行结果：

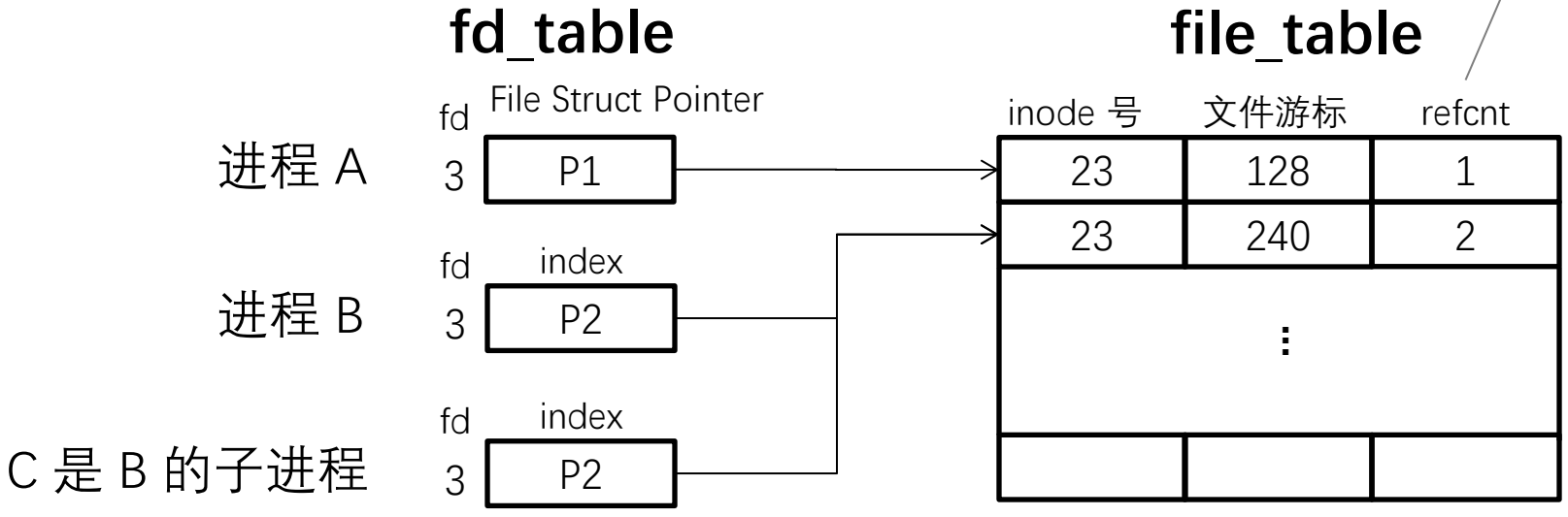
```
➜  os-fs ./a3.out
hello, world
zsxgo, world
➜  os-fs cat file.txt
zsxgo, world
```

注释28行，31行输出 o, world

文件游标共享实例

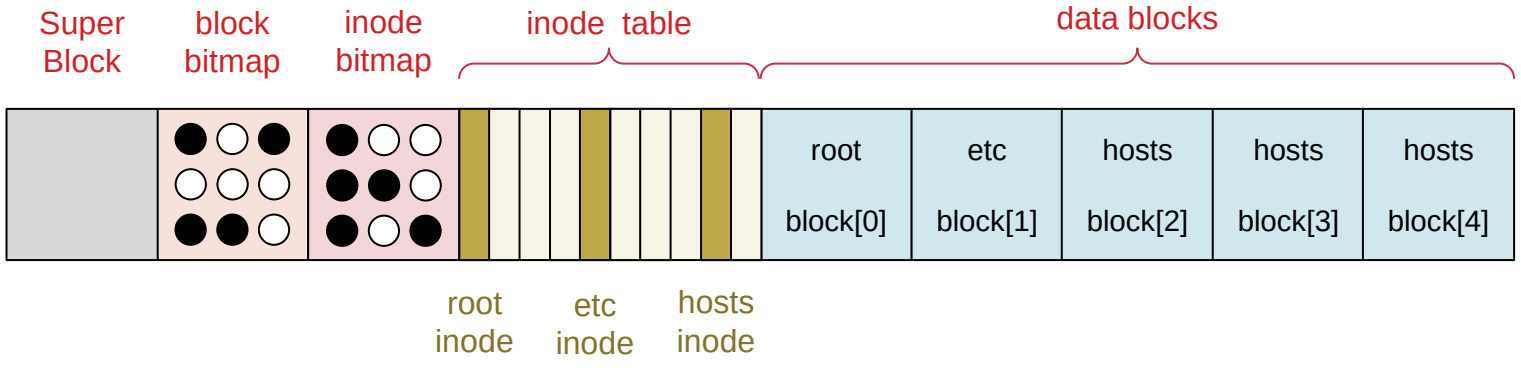
注意：这个refcnt与inode的refcnt不同！

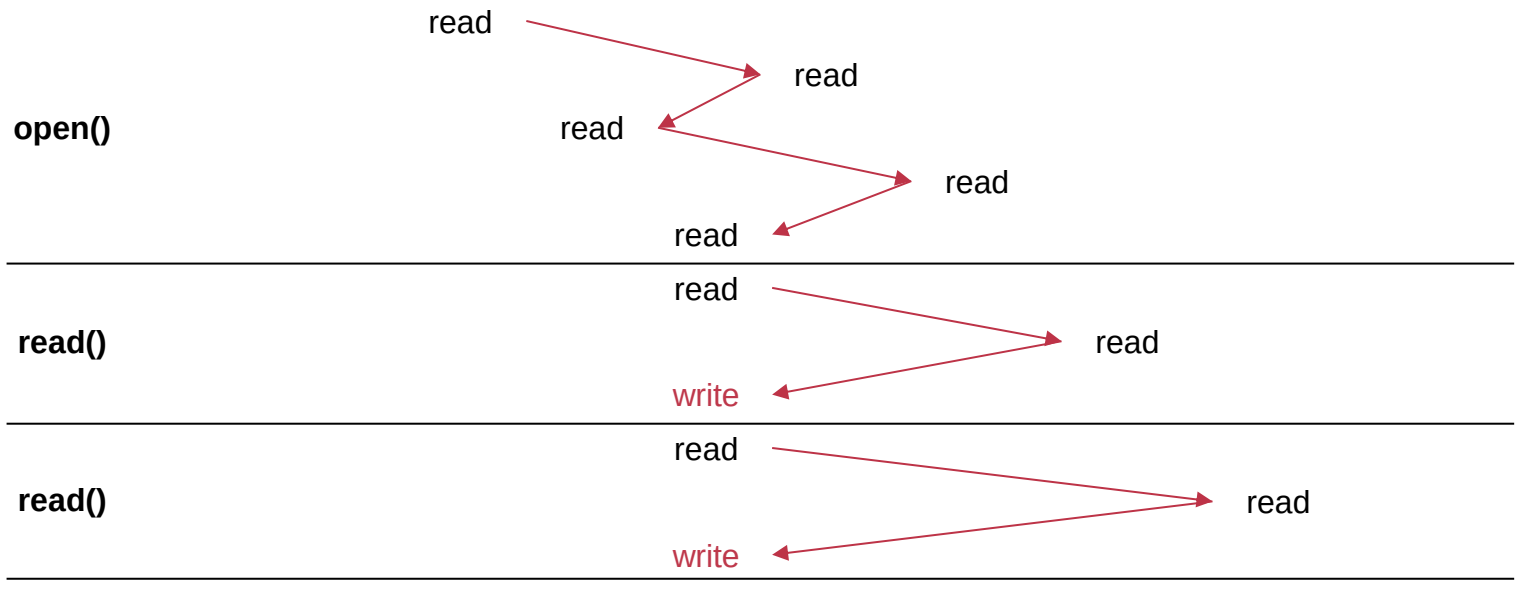
前者：fd引用数量；后者：硬链接数量



- 三个进程 A、B、C 都打开了 inode 号为 23 的文件
- 进程 A 和 B 不共享文件游标
- 进程 B 和 C 共享文件游标

如何实现open/read接口？ 以访问/etc/hosts为例





思考：为什么会有write？ 因为需要更新atime

write 和 close

- **write() 与 read() 类似**
  - 可能需要分配新的 block
  - 更新 inode 的 size 和 mtime
- **close()**
  - 释放 fd_table 中的相关项
  - 减小 file table 中相关项的 refcnt
  - 如果 file table 中相关项 refcnt 为 0，则将其释放

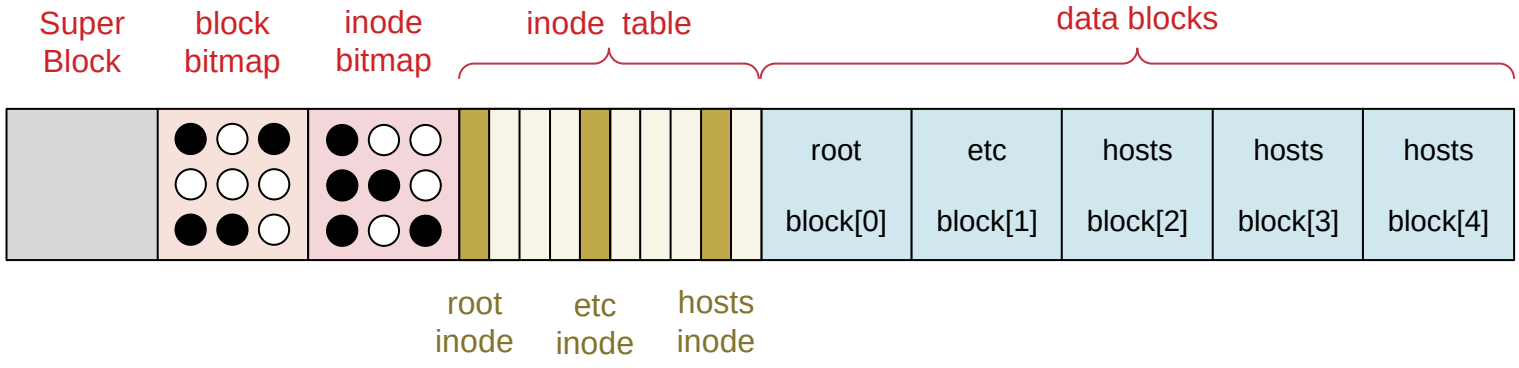
删除一个打开的文件

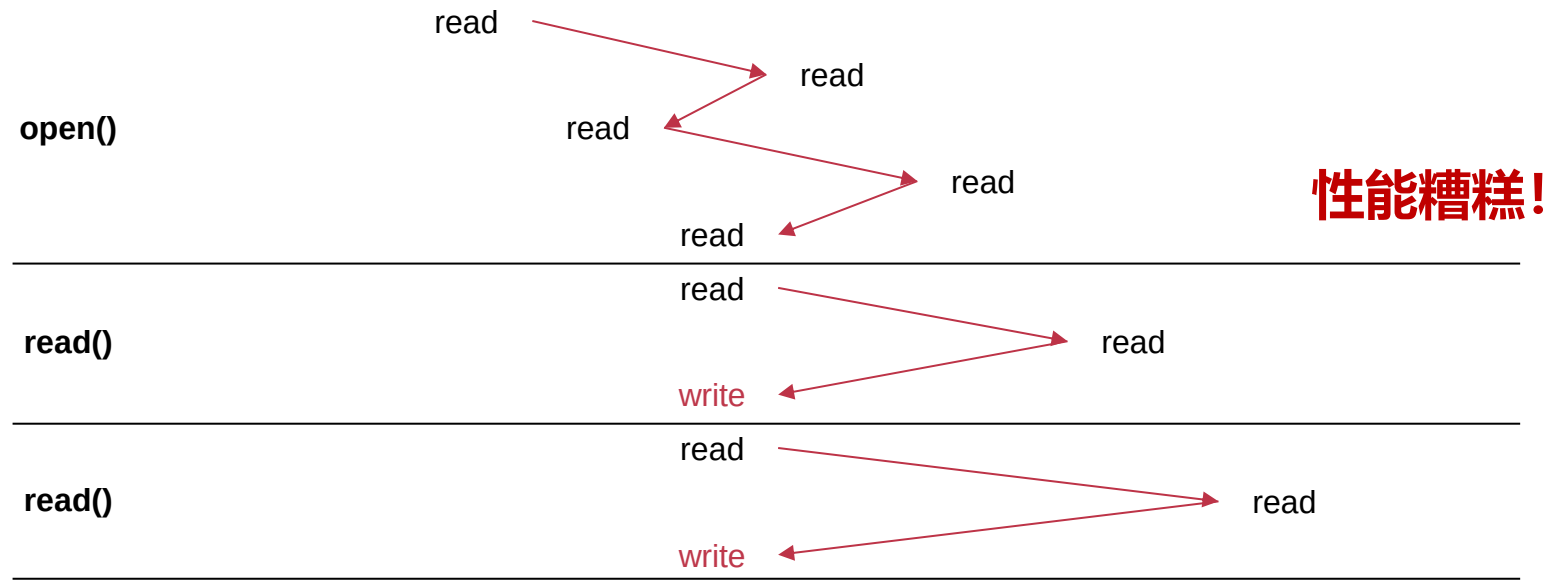
- **进程P1打开了文件 A**
  - 运行 open，在 file_table 和 fd_table 中都增加了一项
- **进程P2将文件 A 删除**
  - 删掉了指向文件 A 的最后一个目录项
  - 文件 A 的 inode 引用数变成了 0
- **文件 A 的 inode 不会被立即释放和删除**
  - 直到前一个进程调用 close 将其关闭
  - （在 Windows 上，则通过"禁止删除打开的文件"实现类似效果）

思考：文件访问的性能

- 一次 OPEN 中有多少磁盘读写?
- 一次 READ 中有多少磁盘读写?

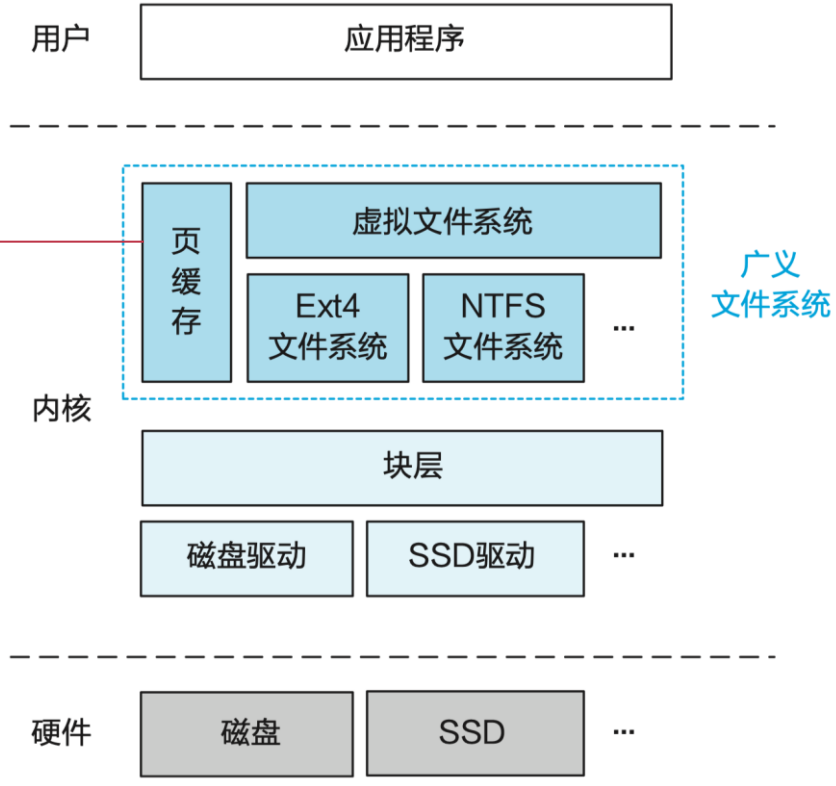
以访问/etc/hosts为例





性能糟糕！

页缓存



性能提升！

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

READ(…)

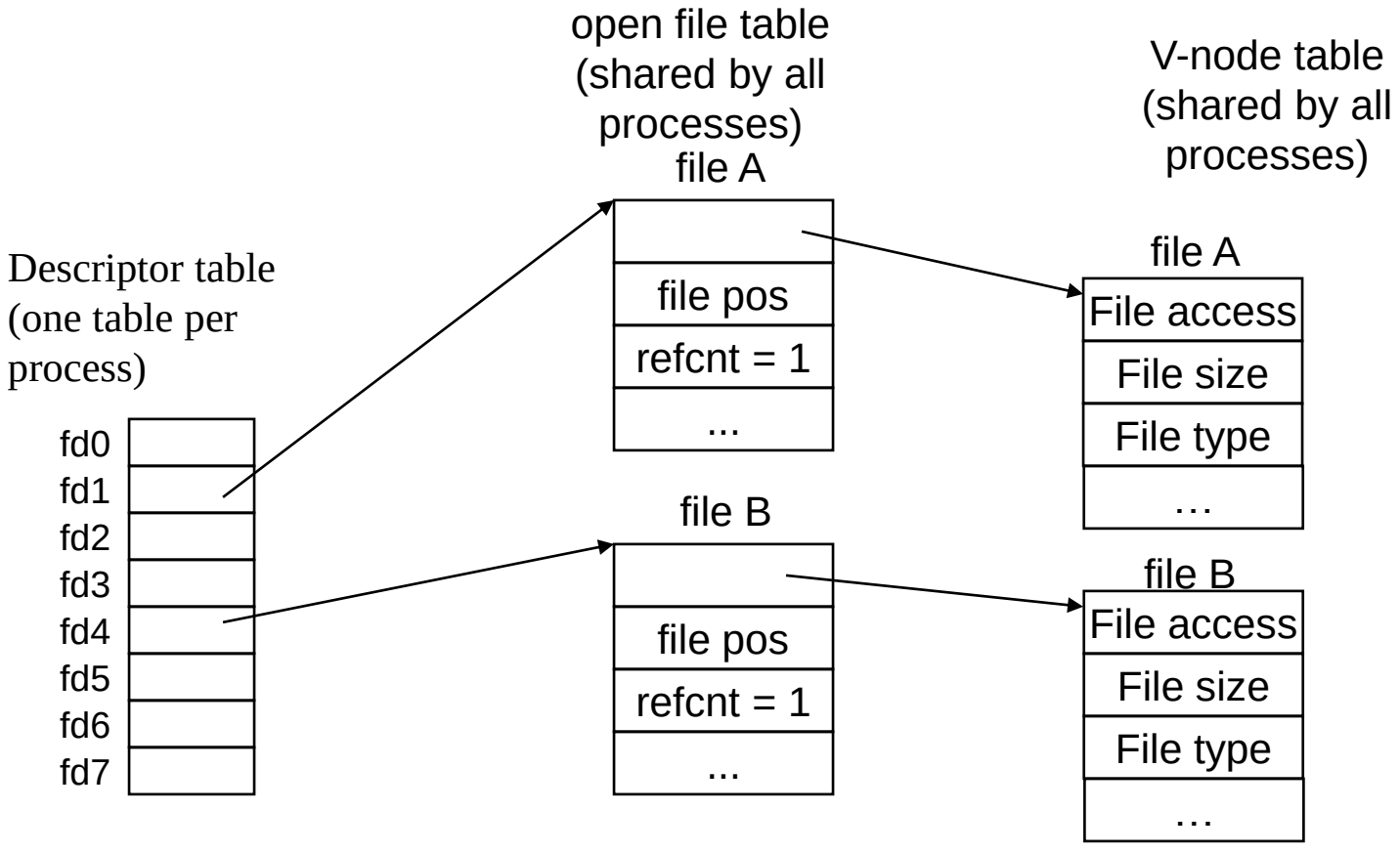
WRITE(…)

…

READ(block-addr, buf)

WRITE(block-addr, buf)

Vnode：缓存inode到内存中



fsync

- **软件/硬件的数据缓存**
  - 缓存了最近被使用的数据块
  - 缓存缺失时，从磁盘中读取
  - 推迟数据向磁盘的写入
  - 寻求机会批量写入，提升性能
  - 问题：如果在写入前发生故障，可能会造成不一致
- **fsync(fd)**
  - 保证对文件的所有修改被写入到存储设备

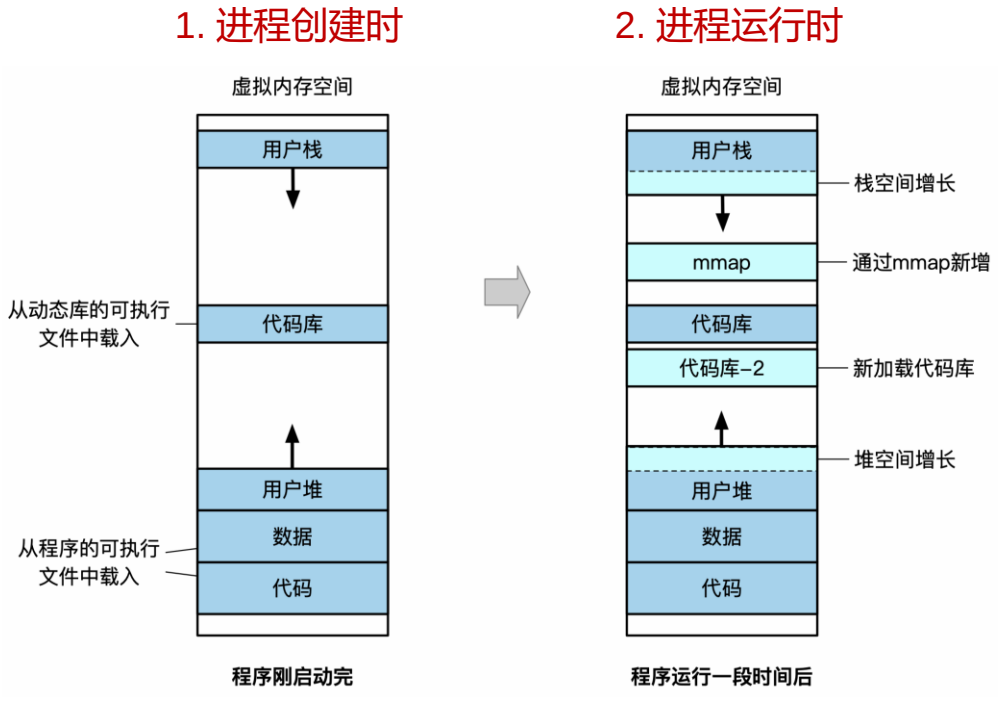
回顾：mmap分配一段虚拟内存区域

- **通常用于把一个文件（或一部分）映射到内存**
  - void *mmap(void *addr,　　// 起始地址
    size_t length,　　// 长度
    int prot,　　// 权限，例如PROT_READ
    int flags,　　// 映射的标志，例如MAP_PRIVATE
    int fd,　　// -1 或者是有效fd
    off_t offset)　　// 偏移，例如从文件的哪里开始映射
    - VMA中还会包含文件映射等信息
- **也可以不映射任何文件，仅仅新建虚拟内存区域（匿名映射）**

回顾：VMA是如何添加的与进程创建

- **途径-1: OS在创建进程时分配**
  - 数据（对应ELF段）
  - 代码（对应ELF段）
  - 栈（初始无内容）

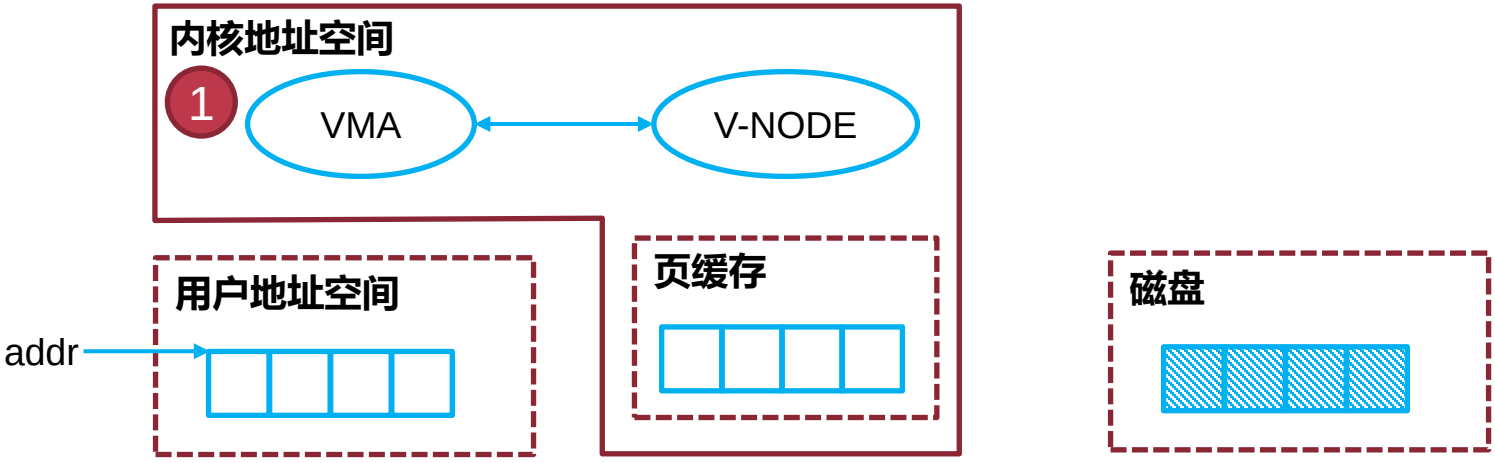
- **途径-2：进程运行时添加**



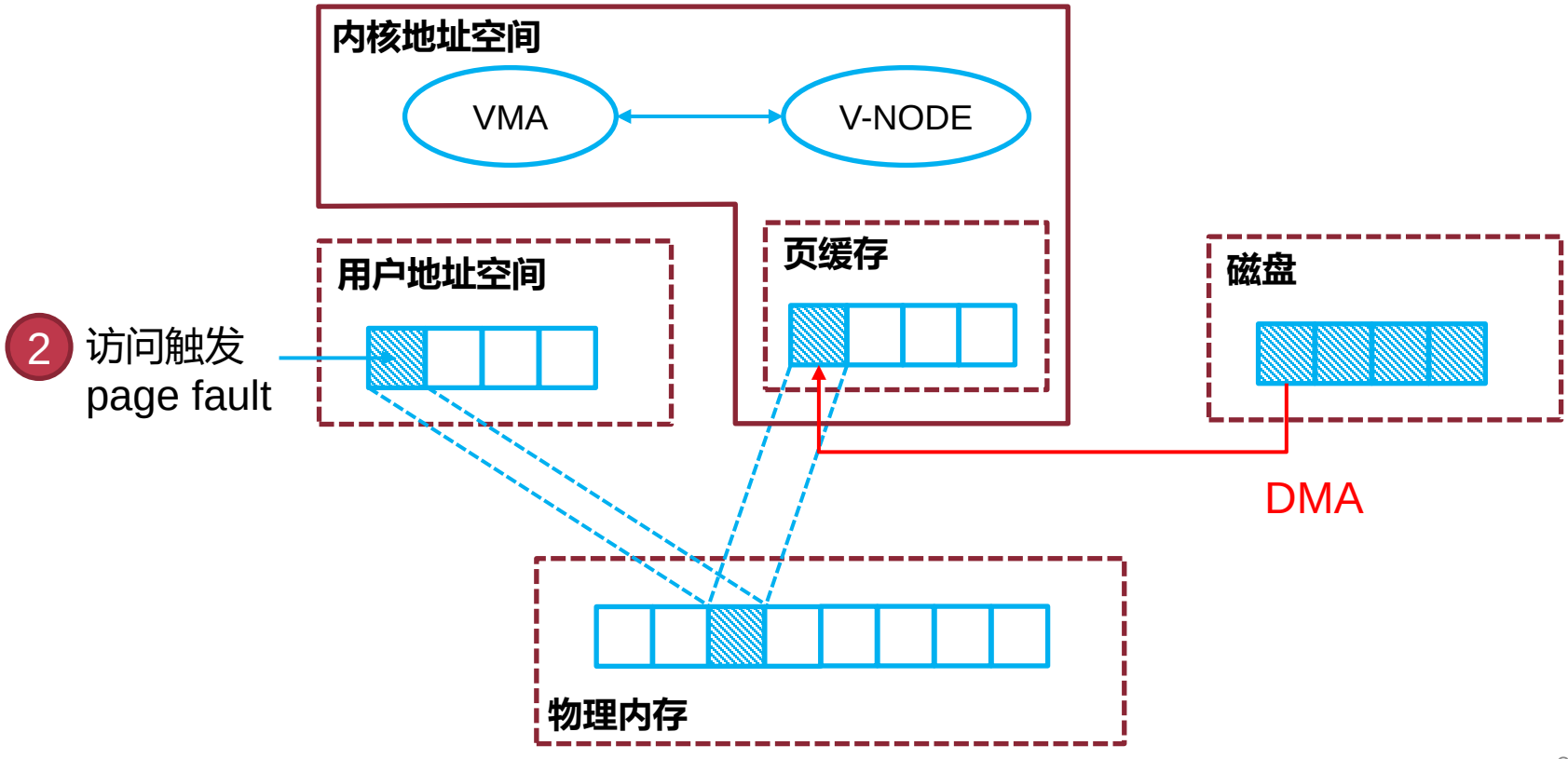
性能对比：mmap vs. read

```
tmac@intel12-pc:~/ieee-ai-os/fs$ time ./read
total read bytes: 5242880000

real    0m1.230s
user    0m0.161s
sys     0m1.070s
tmac@intel12-pc:~/ieee-ai-os/fs$ time ./mmap
Total read 5242880000 bytes

real    0m0.002s
user    0m0.002s
sys     0m0.000s
```

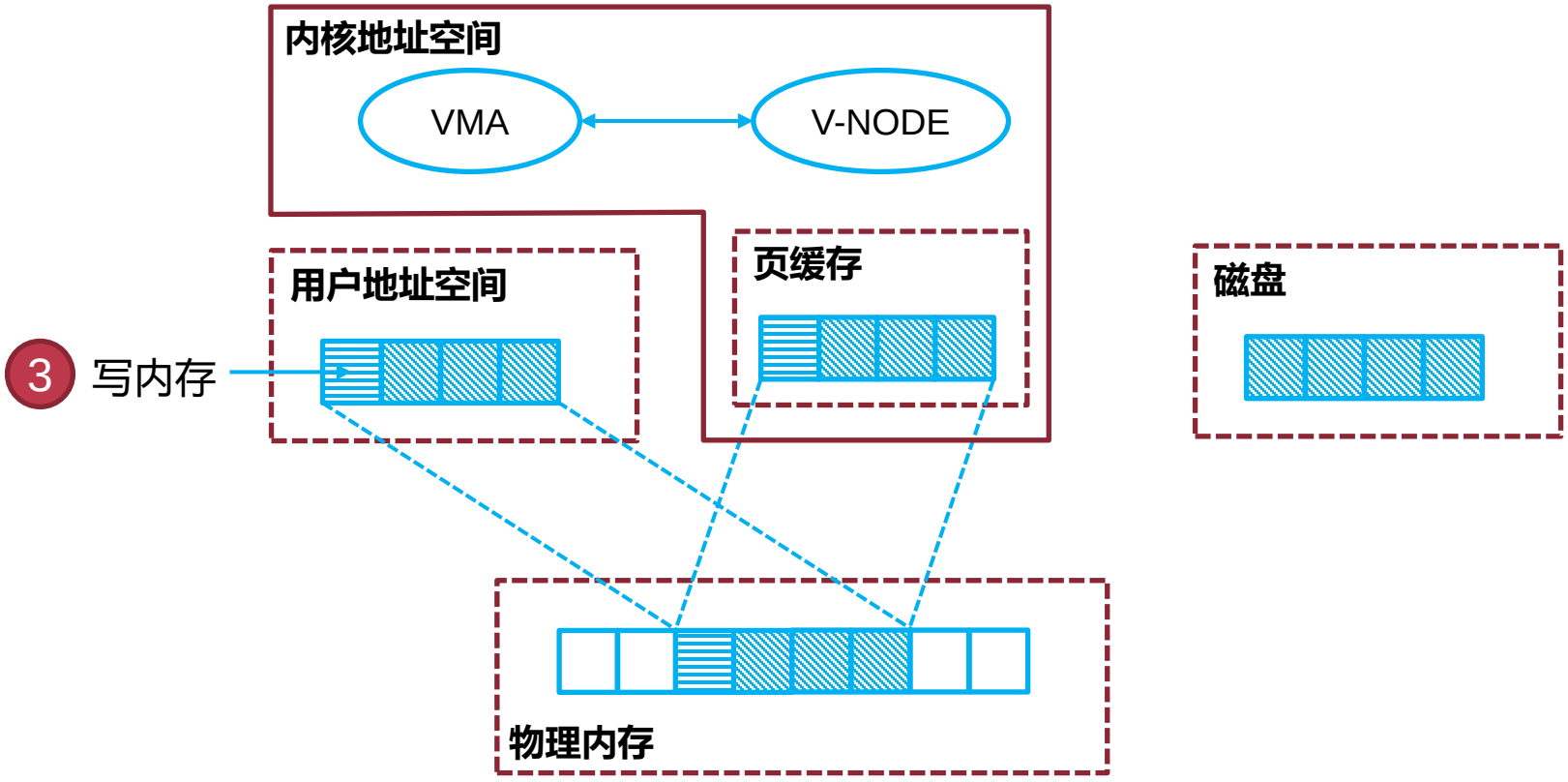
MMAP



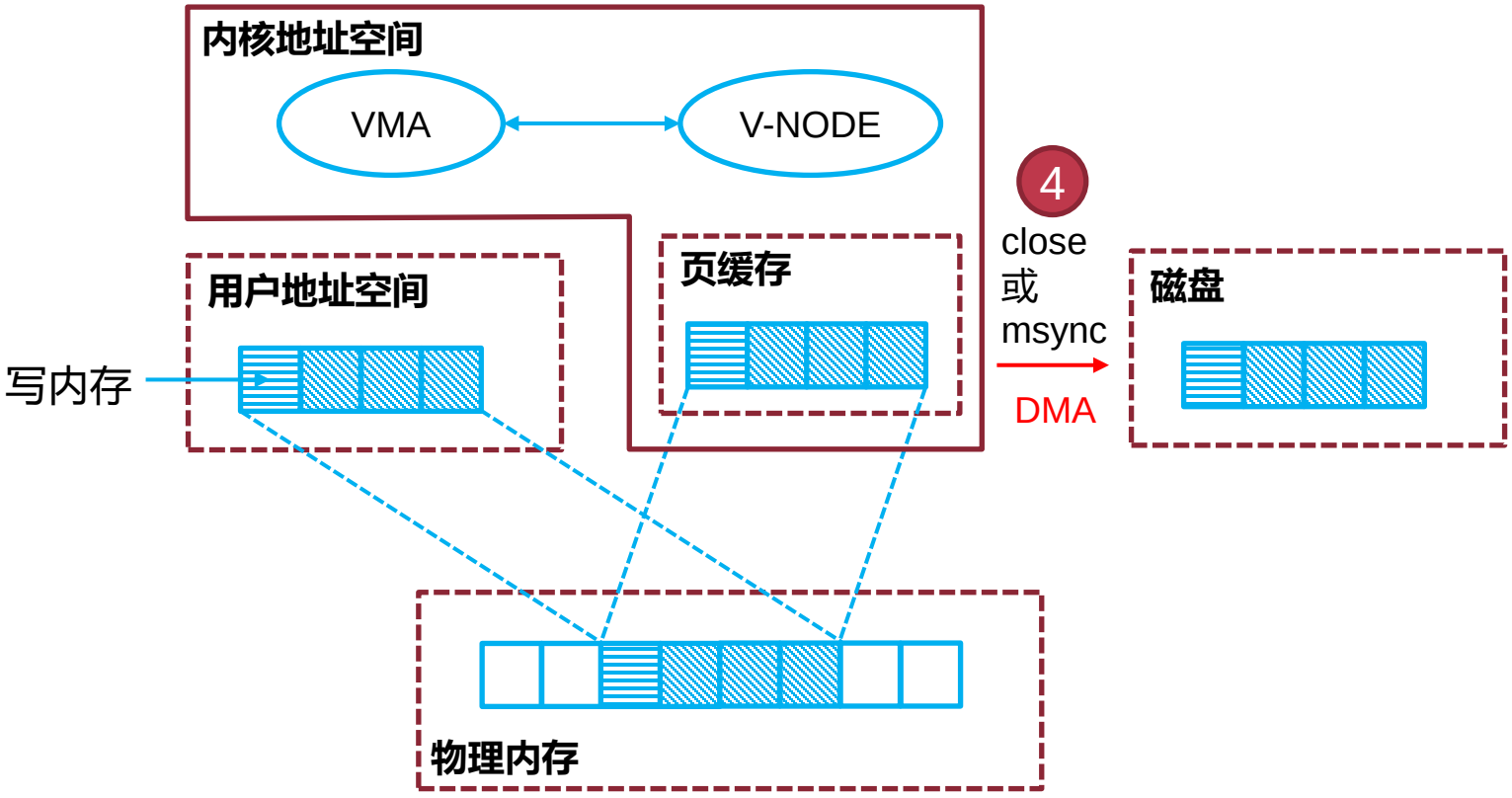
MMAP



MMAP



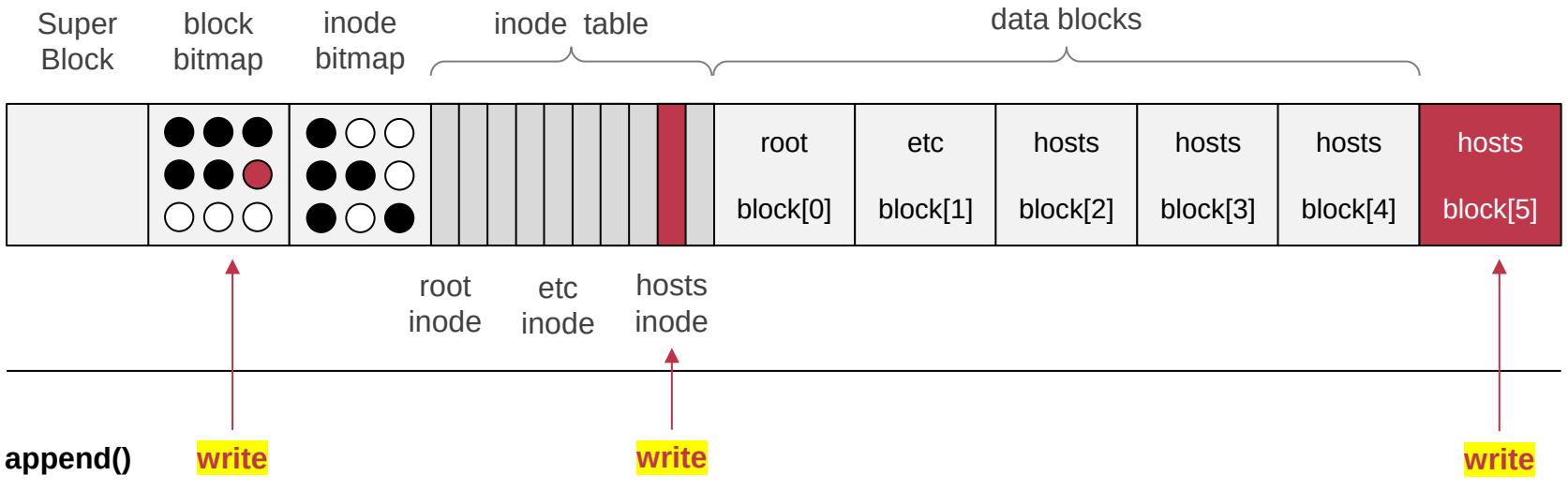
MMAP



文件系统的崩溃一致性

文件系统的崩溃一致性

- **文件系统中保存了多种数据结构**
- **各种数据结构之间存在依赖关系与一致性要求**
  - inode中保存的文件大小，应该与其索引中保存的数据块个数相匹配
  - inode中保存的链接数，应与指向其的目录项个数相同
  - 超级块中保存的文件系统大小，应该与文件系统所管理的空间大小相同
  - 所有inode分配表中标记为空闲的inode均未被使用；标记为已用的inode均可以通过文件系统操作访问
  - ……
- **突发状况（崩溃）可能会造成这些一致性被打破！**



bitmap[5] = 1

inode of /etc/hosts (旧)

- size : 8000
- pointer : block[3]
- pointer : block[4]
- pointer : null

⇨

新的inode

- size : 9000
- pointer : block[3]
- pointer : block[4]
- pointer : block[5]

block[5] =

xxxxxxxxx
xxxxxxxxx
xxxxxxxxx

append包含三个磁盘写

- 更新block位图
- 写入新inode
- 写入新数据

若写入过程发生故障，有6种可能：仅1个写成功（3种），仅2个写成功（3种）
可能的错误：数据错误、空间浪费

崩溃一致性：用户期望

重启并恢复后…

1. 维护文件系统数据结构的内部的不变量

   例如, 没有磁盘块既在free list中也在一个文件中

2. 仅有最近的一些操作没有被保存到磁盘中

   例如：我昨天写的OS Lab的文件还存在

   用户只需要关心最近的几次修改还在不在

3. 没有顺序的异常

   $ echo 99 > result ; echo done > status

方法-1：同步元数据写+fsck

同步元数据写

- 每次元数据写入后，运行sync()保证更新后的元数据入盘

若非正常重启，则运行fsck检查磁盘，具体步骤：

- **1. 检查superblock**
  - 例：保证文件系统大小大于已分配的磁盘块总和
  - 如果出错，则尝试使用superblock的备份
- **2. 检查空闲的block**
  - 扫描所有inode的所有包含的磁盘块
  - 用扫描结果来检验磁盘块的bitmap
  - 对inode bitmap也用类似方法

方法-1：同步元数据写+fsck

- **3. 检查inode的状态**
  - 检查类型：如普通文件、目录、符号链接等
  - 若类型错误，则清除掉inode以及对应的bitmap
- **4. 检查inode链接**
  - 扫描整个文件系统树，核对文件链接的数量
  - 如果某个inode存在但不在任何一个目录，则放到/lost+found
- **5. 检查重复磁盘块**
  - 如：两个inode指向同一个磁盘块
  - 如果一个inode明显有问题则删掉，否则复制磁盘块一边给一个

方法-1：同步元数据写+fsck

- **6. 检查坏的磁盘块ID**
  - 如：指向超出磁盘空间的ID
  - 问：这种情况下，fsck能做什么呢？仅仅是移除这个指针么？
- **7. 检查目录**
  - 这是fsck对数据有更多语义的唯一的一种文件
  - 保证 . 和 .. 是位于头部的目录项
  - 保证目录的链接数只能是1个
  - 保证目录中不会有相同的文件名

方法-1的问题：太慢

- **fsck需要用多长时间?**
  - 对于服务器70GB磁盘（2百万个inode），需要10分钟
  - 时间与磁盘的大小成比例增长

- **同步元数据写导致创建文件等操作非常慢**
  - 例：解压Linux内核源代码需要多久?
    - 创建一个新文件需要8次磁盘写，每次10ms
    - Linux内核大概有6万个源文件
    - 8 x 10ms x 60000 = 1.3小时

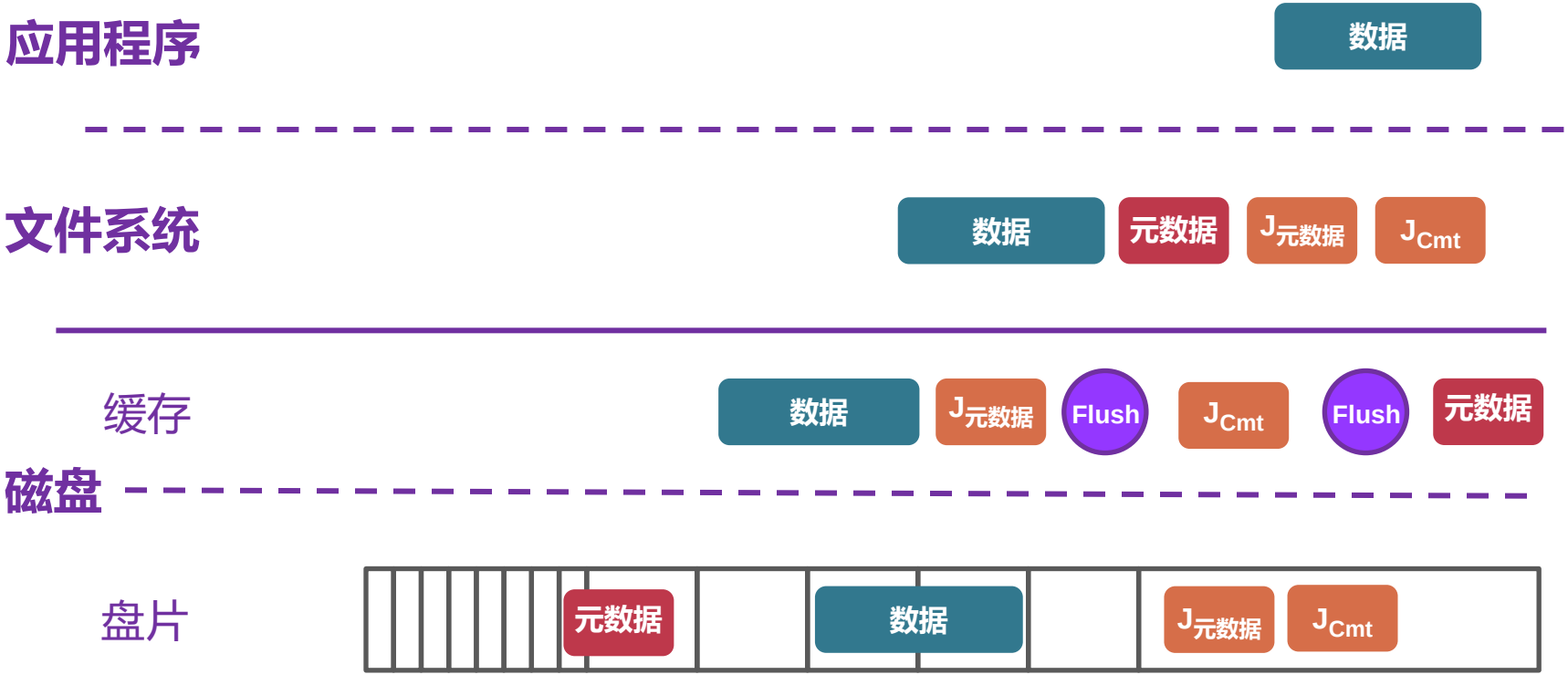
方法-2：日志（Journaling）

- **日志：在磁盘上预留的专门空间**
- **在进行修改之前，先将修改记录到日志中**
  - 如：如何修改block-bitmap、如何修改data
- **所有要进行的修改都记录完毕后，提交日志**
- **确定日志落盘后，再修改数据和元数据**
- **修改完成后，删除日志**

例: Ext4的日志

- **Data mode（即 full journaling）**
  - 数据和元数据都写入日志区域
- **Ordered mode** 默认配置
  - 先写数据（原本的文件位置），再写元数据（日志）
- **Writeback mode**
  - 仅仅将元数据写入日志
  - 数据依然写入原本的位置
  - 日志和数据之间没有顺序保证

Ordered Mode：两次Flush保证顺序



崩溃后，基于日志恢复

- **启动后首先检查日志区域**
  - 若没有任何日志记录，则无需恢复
- **扫描所有已经COMMIT的事务**
  - 若没有COMMIT的事务，则无需恢复
  - 对已经COMMIT的事务，将元数据从日志区写到原本位置
- **完成后清空日志区域**

思考

- Order mode相对Data mode有什么缺点?

- 手机和笔记本电脑等设备有电池/数据中心一般会配有UPS（不间断电源），是否还需要保证文件系统崩溃一致性?

小结

- **三个table关系**
  - fd table
  - file struct table
  - vnode table
- **mmap文件**
- **文件系统日志**