文件系统可靠性

中国科学院大学计算机与控制学院
中国科学院计算技术研究所
2021-12-13

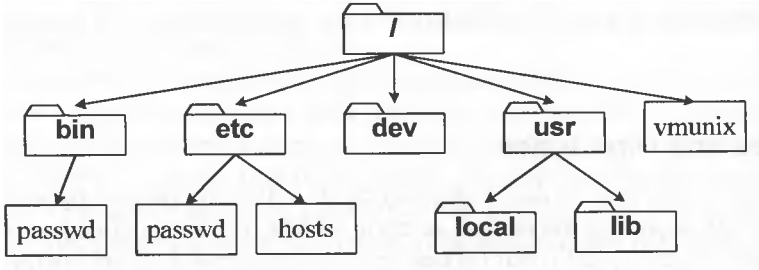
内容提要

- 文件系统可靠性
  - 数据备份
  - 宕机一致性保证
  - 事务与日志文件系统
  - 日志结构文件系统

威胁FS的因素（一）

- FS要求
  - FS给用户提供持久化的数据存储
  - 文件一直要保存完好，除非用户显式删除它们

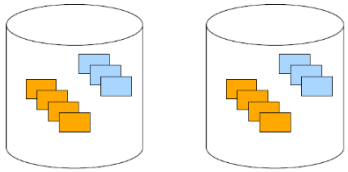
- 威胁一：设备坏
  - 磁盘损坏或磁盘块损坏
  - 超级块：整个FS丢失
  - 位图块、 i-node table
  - 数据块：目录、文件、间址块损坏



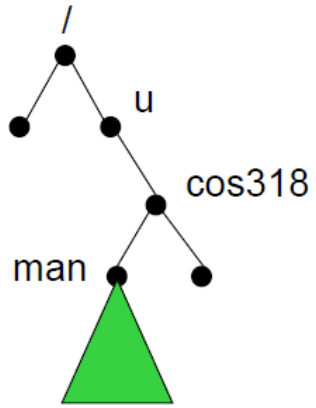
Boot block	Super block	Free space mgmt	I-nodes	Root dir	Files and directories

备份与恢复

- 物理备份与恢复：设备级
  - 将磁盘块逐一拷贝到另一个磁盘上（备份盘）
    - 全复制：原始盘与备份盘在物理上一模一样
    - 增量备份：只拷贝发生变化的块，与上次备份相比
  - 相应的恢复工具



- 逻辑备份与恢复：文件系统级
  - 遍历文件系统目录树，从根目录开始
  - 把你指定的目录和文件拷贝到磁带/光盘/备份磁盘
  - 在备份过程中验证文件系统结构
  - 恢复工具：将你指定的文件或目录树恢复出来
  - 也有两种
    - 全备份：备份整个目录树（即整个FS）
    - 增量备份：只备份发生变化的文件/目录



备份与恢复

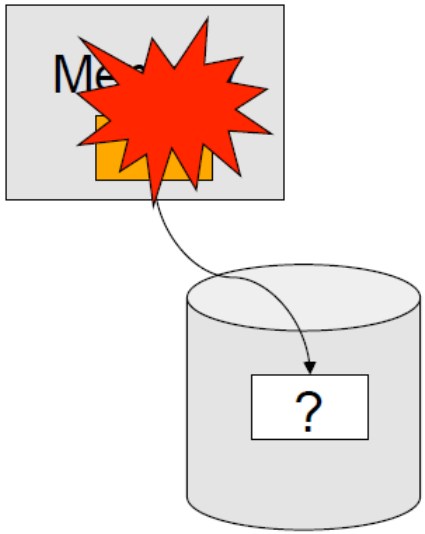
- 物理备份
  - 忽略文件和文件系统结构，处理过程简洁，备份性能高
  - 文件修改时只用备份修改的数据块，不用备份整个文件
  - 不受文件系统限制
- 逻辑备份
  - 备份文件时，文件块可能分散在磁盘上，备份性能受影响
  - 文件修改时需要备份整个文件
  - 受文件系统限制

内容提要

- 文件系统可靠性
  - 数据备份
  - 宕机一致性保证
  - 事务日志文件系统
  - 日志结构文件系统

威胁FS的因素（二）

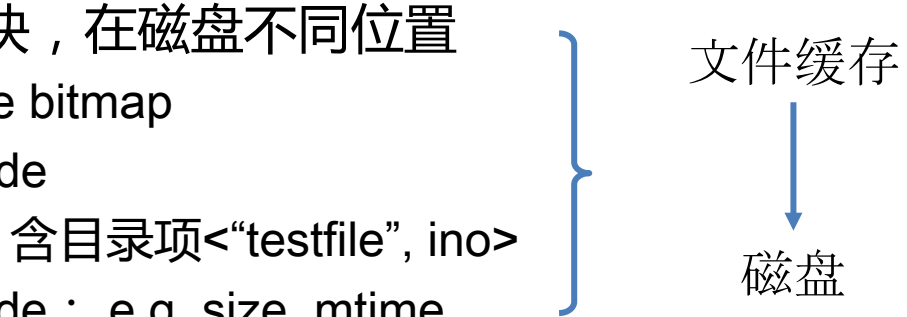
- 威胁二：宕机或掉电
  - 硬件掉电
  - 软件bugs：disk controller、drivers、FS
  - 导致文件缓存中的脏数据没有写回磁盘：目录块、i-node、间址块、数据块

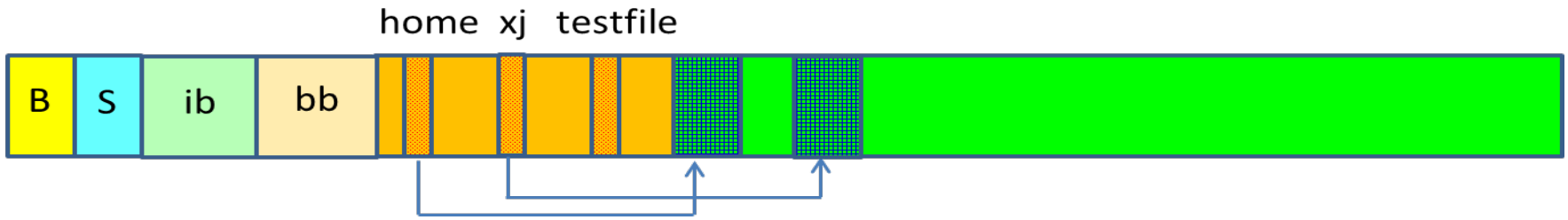


- 挑战
  - 宕机可能发生在任意时刻
  - 性能与可靠性的取舍
    - 宕机使得内存中的数据全部丢失
    - 缓存越多的数据 ➔ 性能越好
    - 缓存越多的数据 ➔ 宕机时丢失的数据越多
  - 一个写操作往往修改多个块
    - 磁盘只能保证原子写一个扇区

宕机的影响

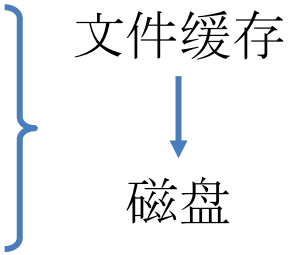
- 创建文件：在当前目录/home/xj下创建文件testfile
  - 需写回4个块，在磁盘不同位置
    - ib: i-node bitmap
    - 文件i-node
    - 目录块：含目录项<“testfile”, ino>
    - 目录i-node： e.g. size, mtime
  - 宕机时，没有全部写回 → **FS不一致**
    - 写回顺序可能是任意的
    - 目录块没写回：有i-node，没有目录项
    - 文件i-node没写回：有目录项，没有i-node
    - …



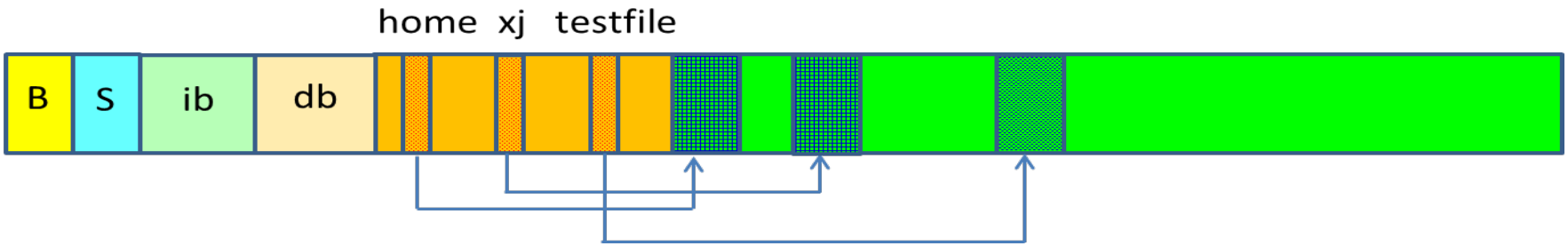


宕机的影响

- 写文件：在/home/xj/testfile末尾写入1个数据块 (Append)
  - 需写回3个块，在磁盘不同位置
    - db: data-block bitmap
    - 文件i-node，e.g. size, mtime, 直接指针
    - 数据块本身



  - 宕机时，没有全部写回 → **FS不一致 & 数据与元数据不一致**
    - 写回顺序可能是任意的
    - **FS不一致**： 文件 i-node 与 bitmap中有一个没写回
    - **数据与元数据不一致**：文件i-node写回，数据块没写回



写回方案一：先写元数据、后写数据

- 写文件：在/home/xj/testfile末尾写入1个数据块 (Append)
  - 路径解析“/home/xj/testfile”

Crash 数据与元数据一致

  - 分配数据块

Crash 数据与元数据一致

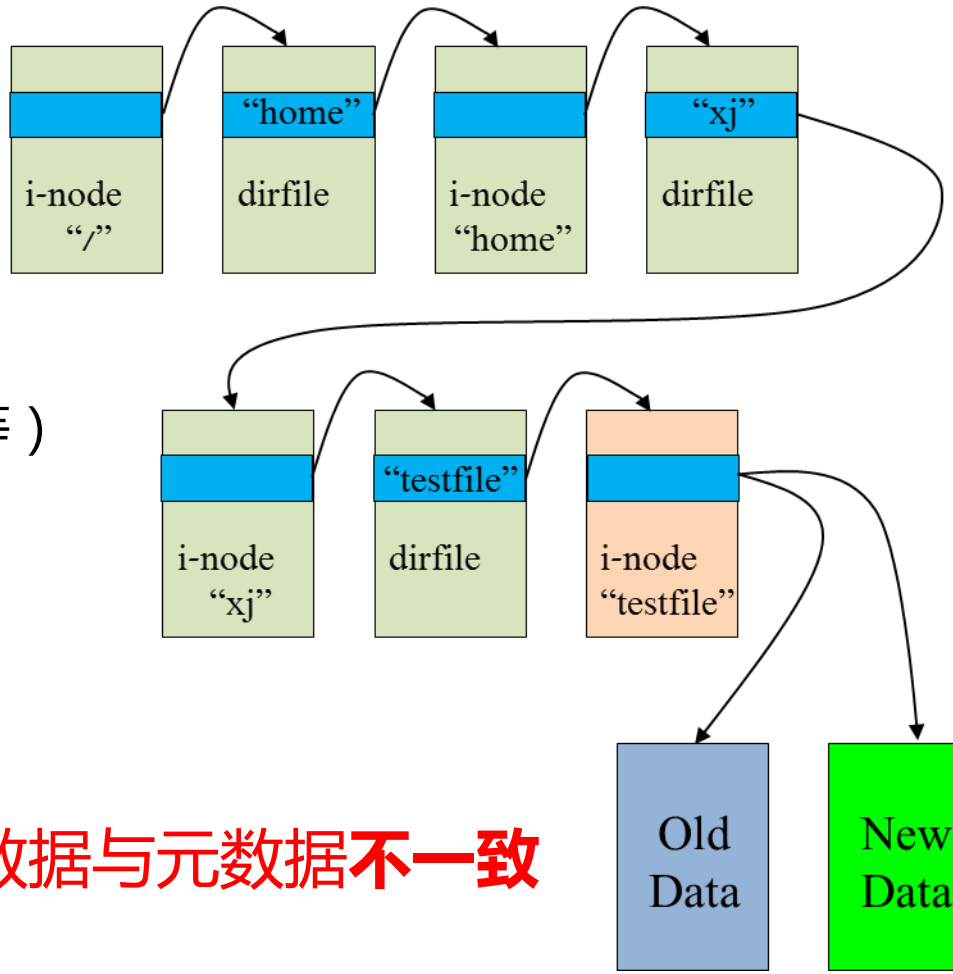
FS不一致，有垃圾块

  - 写inode（直接指针、大小等）

Crash 数据与元数据**不一致**

  - 写新数据块

Crash 数据与元数据一致



先写元数据，宕机后可能出现数据与元数据**不一致**

写回方案二：先写数据、后写元数据

- 写文件：在/home/xj/testfile末尾写入1个数据块 (Append)
  - 路径解析“/home/xj/testfile”

Crash 数据与元数据一致

  - 分配数据块

Crash 数据与元数据一致

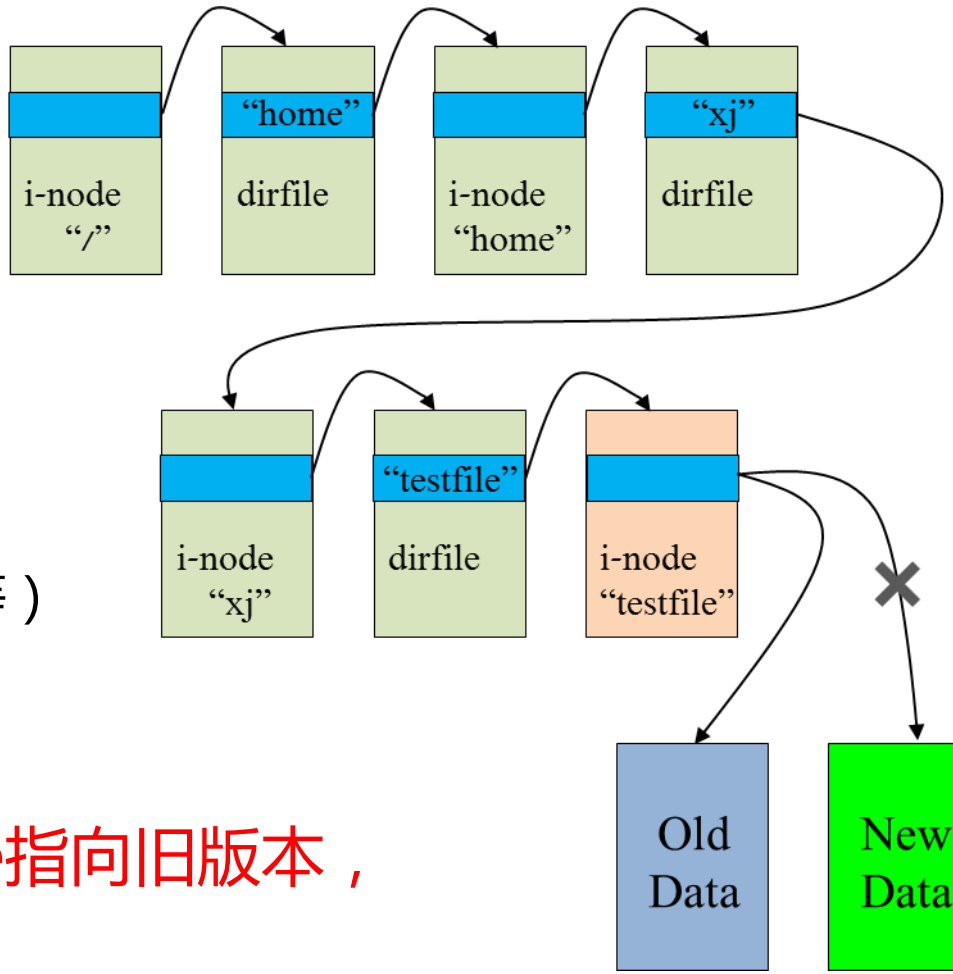
FS不一致，有垃圾块

  - 写新数据块

Crash 数据与元数据一致

  - 写inode（直接指针，大小等）

Crash 数据与元数据一致



先写数据，宕机后可能i-node指向旧版本，
但数据与元数据是**一致的**

宕机一致性保证

- FS一致性
  - 文件系统自身的数据结构（称为FS元数据）一致
    - 超级块、位图、目录项、i-node
  - FS不一致：修改多个元数据的操作
    - 创建/删除文件、创建/删除目录、重命名、硬链接、符号链接、…
    - 写文件
- 数据与元数据一致性
  - 文件元数据 (i-node和间址块)和文件块一致
  - 数据与元数据不一致：写文件
- 数据一致性
  - 一次写多个数据块
  - 数据不一致：写文件
- 宕机一致性保证
  - 一个操作包含多个修改：要么全部写到磁盘，要么都没写到磁盘

宕机一致性保证

- 通用方法：按自底向上顺序进行修改（Ordered Write）
  - 文件的数据块→文件的i-node→目录的数据块→目录i-node…
  - 写回所有的数据块（有文件缓存）
  - 修改文件的i-node，并把它写回磁盘
  - 修改目录块（目录项），并把它写回磁盘
  - 修改目录的i-node，并把它写回磁盘
  - 沿路径向上，直到无修改的目录
- 缺点
  - 虽然保证了数据与元数据的一致性，但在宕机后可能产生垃圾块（FS不一致），可能有数据不一致
  - FS不一致可以通过运行一致性检查工具（例如fsck）清理垃圾块
- 理想境界
  - 保证一致性的修改，而且不留下垃圾
  - 三个一致性保证

fsck: UNIX FS一致性检查工具

- 检查并试图恢复FS的一致性
  - 不能解决所有问题，比如数据与元数据不一致
- 检查superblock
  - 如果fs size < 已分配块，认为它损坏，切换到另一个superblock副本
- 检查块位图
  - 重构已使用块信息：扫描磁盘上所有的i-node和间址块
- 检查i-node位图
  - 重构已使用i-node信息：扫描磁盘上所有目录的目录项
- 检查i-node
  - 通过type域的值（普通文件，目录，符号链接）来判断i-node是否损坏
  - 如果损坏，则清除该i-node及它对应的bitmap位

M. K. McKusick, T.J. Kowalski, **FSCK — The UNIX File System Check Program**, 1985
https://www.usenix.org/legacy/publications/library/proceedings/sd96/sweeney.html

fsck: UNIX FS一致性检查工具

- 检查nlink域
  - 遍历FS的整个目录树，重新计算每个文件的链接数（指向它的目录项个数）
  - 没有目录项指向的i-node，放到 lost+found 目录下
- 检查数据块冲突
  - 是否有两个(或更多)的i-node指向同一数据块
  - 如果有inode损坏，则清理inode，否则把该数据块复制一份
- 检查数据块指针
  - 指针是否越界（>磁盘分区大小）
  - 删除该指针
- 缺点
  - 恢复时间与FS大小成正比
    - 即使只修改了几个块，需要扫描整个磁盘和遍历FS目录树
  - 丢数据： i-node损坏、数据块或间址块损坏

M. K. McKusick, T.J. Kowalski, **FSCK — The UNIX File System Check Program**, 1985
https://www.usenix.org/legacy/publications/library/proceedings/sd96/sweeney.html

内容提要

- 文件系统可靠性
  - 数据备份
  - 宕机一致性保证
  - 事务与日志文件系统
  - 日志结构文件系统

事务概念

- 事务概念来自数据库
  - 交易系统：银行交易、订票系统等
  - 例子：从A账户转账1000元至B账户
- 事务是一组操作，需要具有原子性
  - 要么所有操作都成功完成，要么一个操作也不曾执行过

事务操作接口

- 定义构成事务的一组操作
- 原语
  - Begin Transaction
    - 标记一个事务的开始
  - Commit (End transaction)（提交）
    - 标记一个事务的成功
  - Rollback (Abort transaction)（回滚）
    - 撤消从“Begin transaction”起所发生的所有操作
- 规则
  - 事务可以并发执行
  - 回滚可以在任意时刻执行

事务的实现: Write-Ahead Log（写前日志）

- Begin Transaction
  - 在磁盘上写一条开始日志TxB，标明一个事务开始
- 事务中的修改
  - 所有修改都写日志
  - 事务日志中需要标明事务编号TID
- Commit
  - 在磁盘上写一条结束日志TxE，标明一个事务成功完成
- Checkpoint
  - Commit之后，把该事务中的修改全部写到磁盘上
- 清除日志
  - Checkpoint写完后，清除相应的日志

事务的实现: Write-Ahead Log

- 宕机恢复: replay
  - 如果磁盘上没有结束日志TxE，什么也不做
  - 如果有，按日志重做（redo log），然后清除日志
- 前提假设
  - 写到磁盘上的日志数据都是正确的（错误发现和纠错机制）
  - 宕机后磁盘仍然是好的

例子：原子转账

- 从A账户转帐1000元至B账户

```
BeginTransaction
    A = A – 1000;
    B = B + 1000;
Commit
```

- 步骤

1. **写Begin Trans. 日志**
2. 将A账户的新值写入日志
3. 将B帐户的新值写入日志
4. **写commit日志**
5. 将A的新值写入磁盘
6. 将B的新值写入磁盘
7. **清除日志**

可能的宕机点

- 步骤1之后
- 步骤2之后
- 步骤3之后
- 步骤4之后
- 步骤5之后
- 步骤6之后
- 步骤7之后

内存

A = 4000
B = 1500

磁盘

A = 5000
B = 500

- 讨论
  - 步骤4能与步骤5交换吗?
  - 步骤5能与步骤6交换吗?

日志

TxB	A=4000	B=1500	TxE	

再看事务的实现

- Begin Transaction写TxB日志
- 所有修改都写日志
- Commit写TxE日志
- Checkpoint：Commit之后把修改全部写到磁盘上
- 清除日志: Checkpoint完成后
- 宕机恢复
  - 如果磁盘上没有”commit”日志，什么也不做
  - 如果有，按日志重做，然后清除日志

如果在恢复的过程中再发生宕机，会怎么样？

再看事务的实现

- Begin Transaction写TxB日志
- 所有修改都写日志
- Commit写TxE日志
- Checkpoint：Commit之后把修改全部写到磁盘上
- 清除日志: Checkpoint完成后
- 宕机恢复
  - 如果磁盘上没有”commit”日志，什么也不做
  - 如果有，按日志重做，然后清除日志
- 前提
  - 日志中记录的所有修改必须是幂等的
  - 每个事务有唯一的编号TID
  - 必须有办法确认写磁盘完成

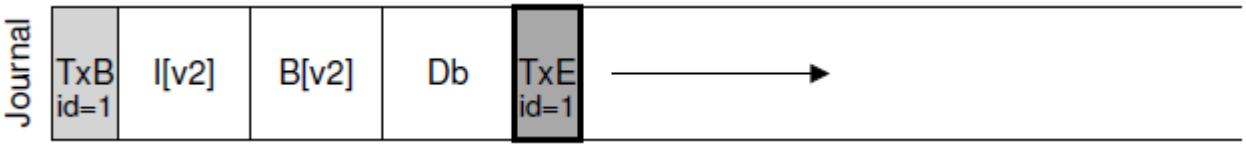
将事务用于FS

- 日志文件系统（journaling file system）
  - 用事务来实现文件操作
  - 每个文件操作都作为一个事务
    - 创建/删除文件、创建/删除目录、重命名、硬链接、符号链接、…
    - 写文件
  - 第一个日志文件系统：Cedar FS [1987]
  - 很多商用文件系统：e.g. NTFS, JFS, XFS, Linux ext2/3/4, …
- 宕机恢复
  - 按日志重做一遍
  - 简单、高效
    - 恢复时间与日志大小成正比，秒级
  - 日志必须是等幂的
  - 这样就不再需要fsck吗？

日志文件系统: 数据日志 (data journaling)

- 记录所有修改的日志：数据 & 元数据
- 例：在一个文件末尾追加一个数据块
  - 修改文件的i-node、修改bitmap、写数据块
- 流程
  - 写日志：TxB、i-node日志、bitmap日志、数据块日志
  - 提交日志commit：写TxE

BeginTransaction
修改 i-node
修改 bitmap
写数据块
Commit

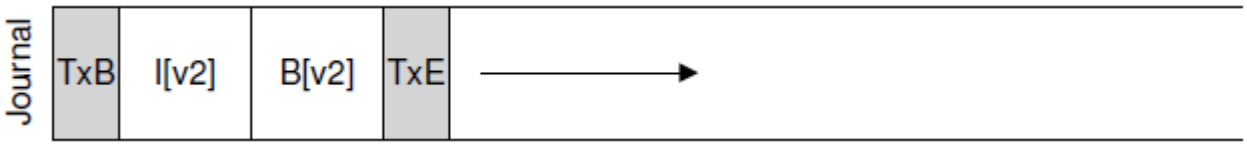


  - Checkpoint: 修改磁盘上的i-node、bitmap、数据块
  - 清除日志

- 日志开销
  - 所有数据块写两次磁盘

日志文件系统：元数据日志 (metadata journaling)

- 只记录元数据修改的日志
- 例：在一个文件末尾追加一个数据块
  - 修改文件的i-node、修改bitmap、写数据块
- 流程
  - 写数据块
  - 写日志：TxB、i-node日志、bitmap日志
  - 提交日志commit：写TxE



  - Checkpoint: 修改磁盘上的i-node、bitmap
  - 清除日志

- 日志开销
  - 所有数据块只写一次磁盘

写数据块
BeginTransaction
修改 i-node
修改 bitmap
Commit

只记录元数据日志，
会有问题吗？

日志文件系统

- 性能问题
  - 增加额外的写磁盘开销：每个日志都要同步写磁盘
  - 写放大
    - 即使只修改一个块中少量内容（十几字节），也需要写日志
    - Bitmap一个bit修改，也需要对bitmap block写日志
- 改进办法
  - 批量写日志：以牺牲可靠性换取性能
    - 宕机仍然可能丢数据，但不会损坏FS一致性
  - 用NVRAM（non-volatile RAM）来保存日志
- 可靠性
  - 无法应对硬件故障，比如磁盘扇区坏

日志管理

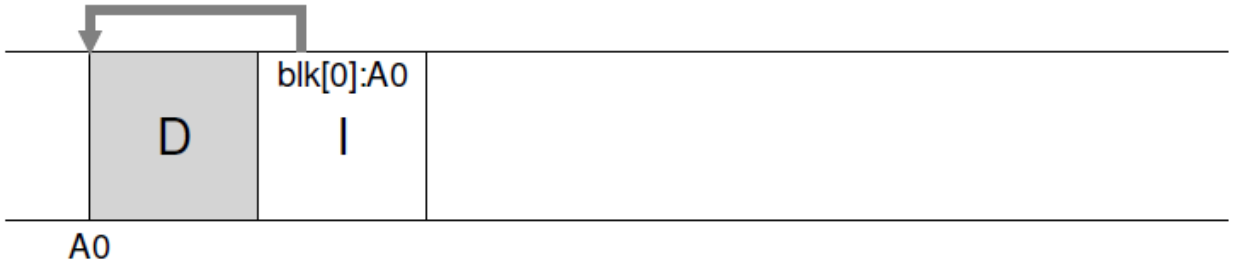
- 需要多大的日志？
  - 日志只在宕机恢复时需要
  - 日志过小，很快占满日志空间，无法接受新日志
  - 日志过大，导致恢复时间长
- 方法
  - 固定大小，顺序地、Append-only写，循环使用 (circular log)
  - 定期做checkpoint：把缓存里的修改内容刷回磁盘
  - checkpoint之后，可以释放日志所占空间

内容提要

- 文件系统可靠性
  - 数据备份
  - 宕机一致性保证
  - 事务与日志文件系统
  - 日志结构文件系统

LFS: Log-Structured File System

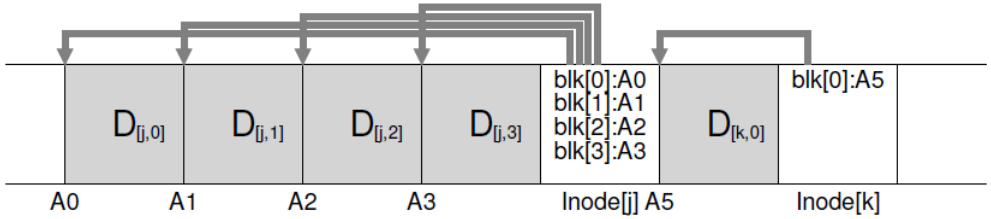
- 思想
  - 像写日志那样顺序地写磁盘
- 具体机制
  - 每次写文件块写到新位置(日志末尾): out-of-place update（ vs. in-place update ）
  - 不需要bitmap来管理空闲空间
  - 文件块采用多级索引 (同FFS): 文件块位置记录在i-node中
  - 每次写文件采用一致性修改: 先写文件块，再写i-node
- 例子：在文件foo中写一个数据块，假设它的ino是k



小粒度写不能发挥磁盘带宽

大粒度顺序写

- Segment: 大粒度的内存buffer
  - 缓存多个写, 一次把整个segment写到磁盘
- 例子：Segment缓存了两组写
  - 向一个文件写4块，向另一个文件写1块

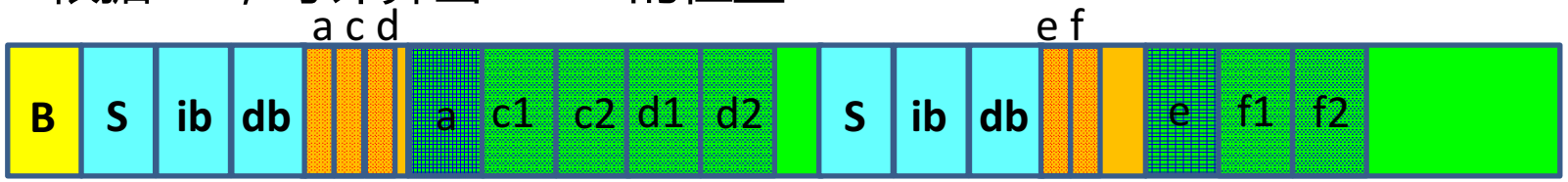


Segment多大才合适？

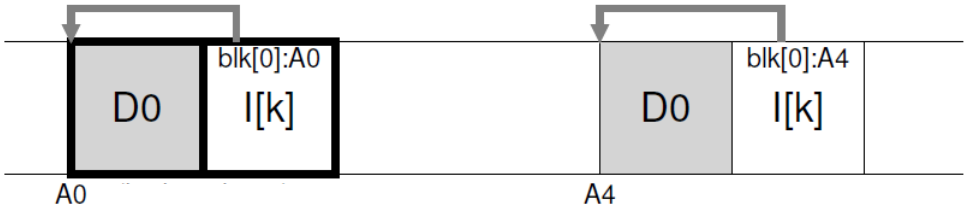
Block Size	Transfer Bandwidth	% of Disk Transfer Bandwidth
1 KB	111KB/s	1‰
4 KB	442KB/s	4‰
10 KB	1MB/s	1%
100 KB	10 MB/s	10%
0.9 MB	50 MB/s	50%
8.1 MB	90 MB/s	90%

怎么读i-node？

- UFS和FFS
  - 根据ino，可计算出i-node的位置



- LFS
  - 每次写文件块，都要写i-node
  - 每次写到新位置
  - 一个文件的i-node在磁盘没有固定位置

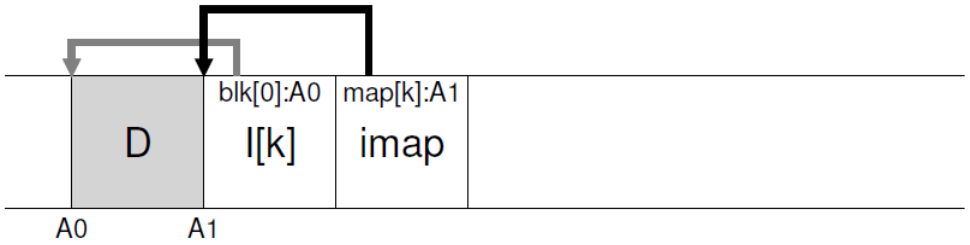


1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

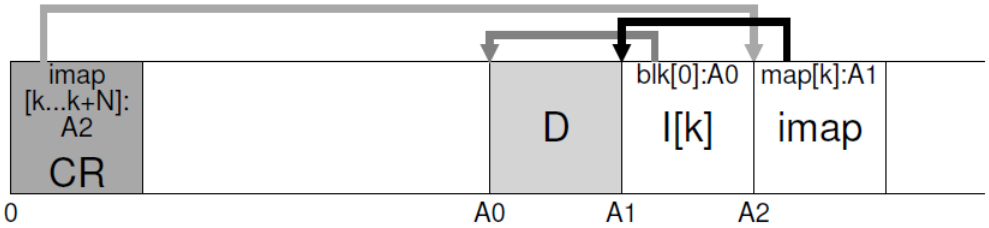
把i-node的位置记录在目录项中可以吗？

imap

- 记录ino→i-node磁盘地址: 只记录最新的i-node地址
- 怎么存？
  - 磁盘上的固定位置
  - 每次写文件，都要修改imap
  - 写日志与写imap需要长距离寻道，性能比FFS还差
- LFS的方法
  - imap块随文件块和i-node一起写到日志中

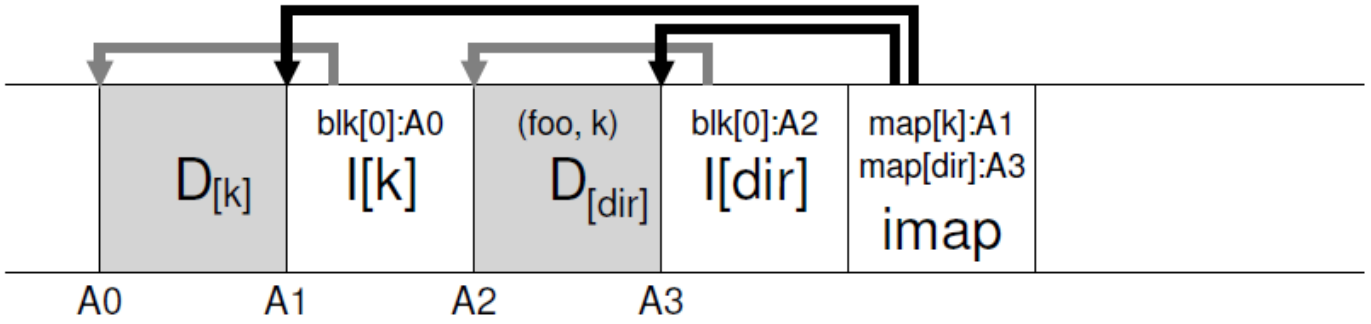


  - **CR(Checkpoint Region)**记录每个imap块的磁盘地址 (最新)
  - CR位于磁盘上固定位置：有两个CR，分别在磁盘头和尾



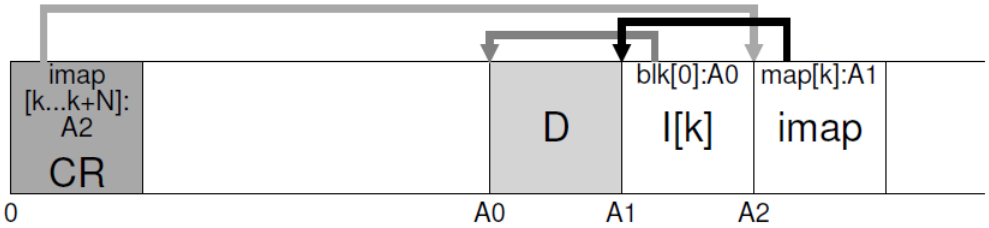
关于目录

- 目录采用与文件一样的方式来写
- 例子：创建文件/home/os/foo，并写入1块



读文件

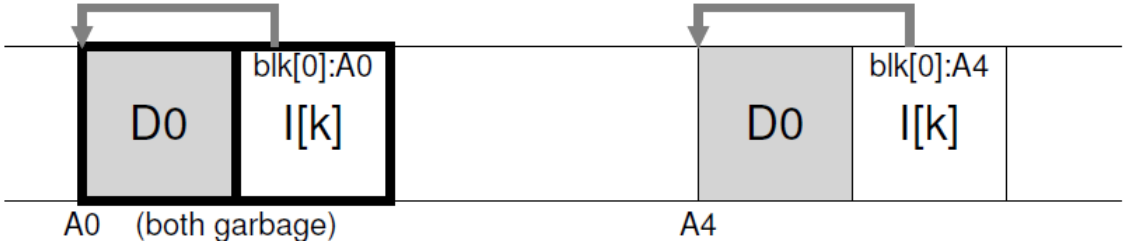
- 假设LFS刚挂载，内存里什么也没有



  - 先读CR，把CR缓存在内存，以后就不用读了
  - 根据ino，知道它所在的imap块
  - 查CR, 得到imap块的磁盘地址
  - 读imap块, 得到ino对应的i-node的磁盘地址
  - 读i-node, 查文件块索引，得到文件块的磁盘地址
  - 读文件块

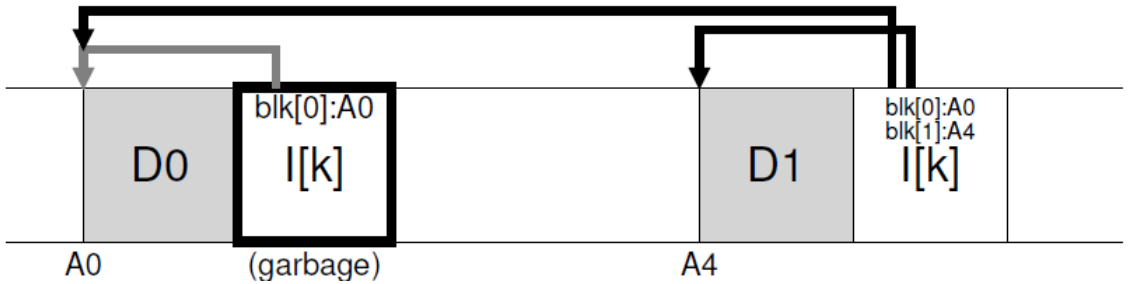
修改文件

- 例1：修改/home/os/foo的第一块



  - 原来的数据块变为无效→垃圾

- 例2：在/home/os/foo末尾追加写一块

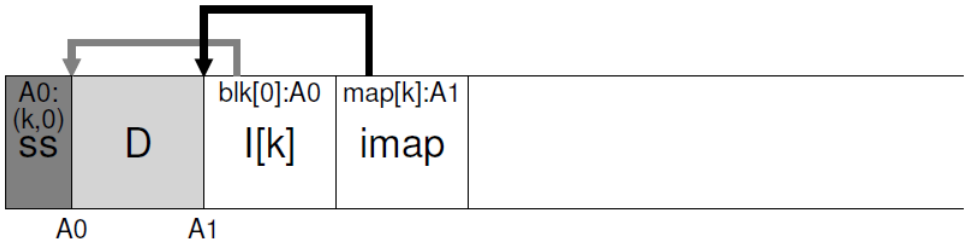


  - 原来的i-node变为无效→垃圾

overwrite(覆盖写)产生垃圾

垃圾回收

- 原理
  - 后台进程cleaner周期性地检查一定数量的segment
  - 把每个segment中的活块(live)拷贝到新的segment中
  - M个segment的活块占据N个新segment (N<M)
- 怎么知道块是不是活的 (live)？
  - Segment summary block
    - 记录每个数据块的ino和文件内偏移
    - 位于segment头部



  - 通过查summary block、imap和文件索引可得到当前这块的地址

垃圾回收

- 何时回收
  - 周期性回收
  - 空闲时：无访问或访问少
  - 磁盘快满时
- 回收什么样的segment
  - 热segment: 块频繁被重写
  - 冷segment: 部分死块，部分稳定块 (不重写)
  - 优先回收冷segment，推迟回收热segment（直至该segment里的数据块都被重写过）

宕机恢复

- LFS的最新修改在日志末尾
- CR记录第一个segment和最后一个segment地址
  - 每个segment指向下一个segment
- CR更新
  - 写CR第一个块，带时间戳
  - 写CR本身内容
  - 写CR最后一个块，带时间戳
  - 周期性将CR刷盘（例如，每30s）
  - 两个CR（磁盘头和尾）交替写，减少CR更新时宕机的影响
- CR一致性保证
  - 第一个块和最后一个块的时间戳一致

宕机恢复

- 恢复方式
  - 使用最后一次完成且一致的Checkpoint Region
    - 时间戳最新的&完整的CR
    - 可以获得最后一次checkpoint时的CR内容，包括imap等数据结构的地址
  - CR周期性刷盘，此时CR内容可能已过时，即最后一次checkpoint后的磁盘写没有被恢复
- 恢复优化（回滚）
  - 使用最后一次修改的CR（不一定是一致的）重构最新的文件修改
    - 根据CR找到日志末尾（有记录），检查后续写的segment
    - 将其中有效的更新恢复到文件系统，例如，将segment summary block中记录的inode更新到imap中
- 恢复快
  - 无需fsck，无需扫描磁盘
  - 秒级 vs. 小时级

总结

- 宕机一致性
  - FS一致性 vs. 数据与元数据一致性 vs. 数据一致性
    - 有序写保证数据和元数据一致性
    - fsck检查FS一致性
  - 事务和原子性保证
  - 日志 (write-ahead log) 与 重做 (replay)
    - 记日志、提交commit 与 Checkpoint
  - 日志文件系统
    - 数据日志与元数据日志

总结

- LFS目标：提高写性能
- 思想：试图消除对磁盘的小粒度随机写和同步写
- 方法：像写日志一样大粒度顺序写磁盘
  - 每次写到日志末尾
  - 需要垃圾回收
  - 读性能受影响
- 区别
  - 不要与日志文件系统混淆
  - 不要与事务的日志混淆
- out-of-place update影响力
  - out-of-place update FS: WAFL、ZFS、Btrfs
  - out-of-place update B-tree: LSM Tree，Google LevelDB、Facebook RocksDB
  - SSD的闪存控制器（FTL）