Persistence (II)
File System - 3
(OSTEP Section 42)

Outline

- Linux I/O Stack
- File System
- File System Implementation
- Fast File System
- FSCK and Journaling
- Log-structured File Systems （不讲）
- Data Integrity and Protection （不讲）

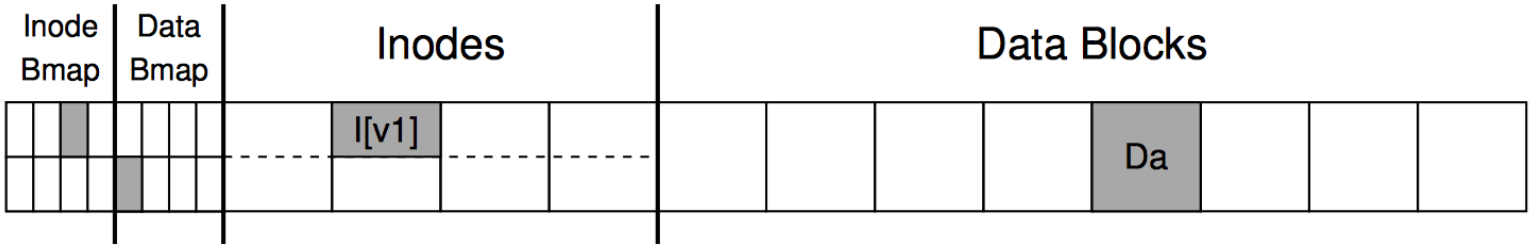
FSCK and Journaling

- 可容忍的系统状态：
  - 完全正常、完全失效（e.g, RAM）
  - 部分失效【存储/数据库、分布式系统】
- **Crash-consistency problem**
  - 例如连续 2 次写入高度关联的 A 和 B，在中间发生 crash
    - 原子性问题
  - 系统进入一个 inconsistent 状态
  - 解决方法
    - **FSCK: file system checker**
    - **Journaling: write-ahead log**

FSCK and Journaling

- 一个例子
  - 一个文件大小为 1，只包括一个数据块
    - inode 部分域:

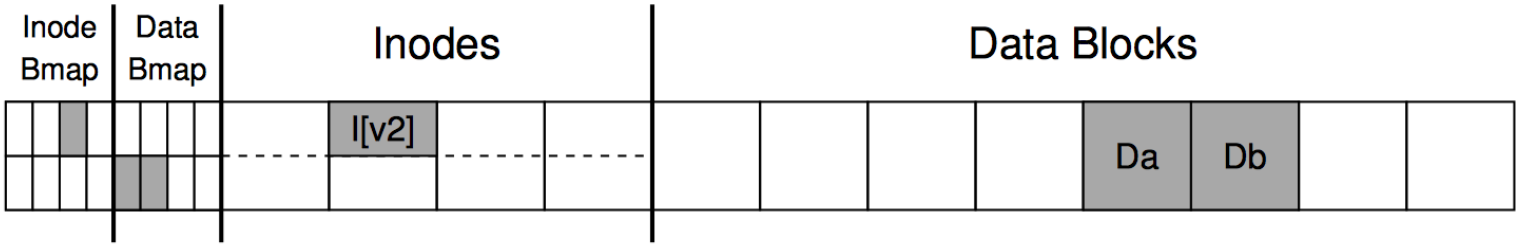
```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```



FSCK and Journaling

- 一个例子
  - 后面文件又追加写入一个 **block Db**
  - **inode** 部分域:

```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```



FSCK and Journaling

- 三个具体的写操作
  - Bitmap
  - Inode
  - Data block (Db)
- 先写入 page cache/buffer cache，延迟写入磁盘
  - 加入中间 crash，有多种可能：

有几种出错的可能?

FSCK and Journaling

- **#1. Just the data block (Db) is written to disk**
  - **db 内容丢失 (lost update)，但不会造成其他问题**
- **#2. Just the updated inode (I[v2]) is written to disk**
  - **inode 与 bitmap 不一致，且 inode 指向 dirty db**
- **#3. Just the updated bitmap (B[v2]) is written to disk**
  - **inode 与 bitmap 不一致，造成 space leak**

FSCK and Journaling

- **#4. inode (I[v2]) and bitmap (B[v2]) written, but not data (Db)**
  - 元数据一致，但是数据是垃圾数据
- **#5. inode (I[v2]) and data block (Db) written, but not the bitmap (B[v2])**
  - 读取正确，但数据可能被覆盖
- **#6. bitmap (B[v2]) and data block (Db) written, but not the inode (I[v2])**
  - 元数据不一致

FSCK and Journaling

- **Solution #1: The File System Checker**
  - **FSCK 最大的问题:**
    - **整盘扫描，太慢了!**
    - **是一种补救措施，只能修复部分问题**

FSCK and Journaling

- **Solution #2: Journaling (Write-Ahead Logging)**
  - **WAL 来自于 database 领域**
  - **最早使用 journaling 的文件系统是 Cedar(1987)**
  - **现在广范使用： Linux ext3, ext4, reiserfs, JFS, XFS, NTFS**

journal(ing) == log(ging)
有时候不同领域用同一个词描述不同的东西
有时候不同领域用不同的词描述同一个东西

Journaling

- **Solution #2: Journaling (Write-Ahead Logging)**
  - 将上层用户对 FS 的操作看做一个 transaction (tx)
    - 例如向某文件中追加一个 data block (db)
  - 一个 tx 中可能包含多个对 FS 内部数据的写入
    - 例如写入 inode(I[v2]), bitmap(B[v2]), 和 Db
  - 在写数据前，先写 log，描述要做什么

Journaling

- **Solution #2: Journaling (Write-Ahead Logging)**
  - 当写 log 完成时，即可告知用户操作完成
  - 如果写 log 过程中系统崩溃，恢复时则无视可能不完整的 log，就当 tx 没发生
  - 如果写数据过程中系统崩溃，恢复时可以依据 log 的内容继续完成数据写入

Journaling

- **Solution #2: Journaling (Write-Ahead Logging)**
  - 如果某个时间点之前的全部 tx 都已经完成对 FS 的修改，则清除该时间点之前的 log 并生成 checkpoint
    - 避免 log 无限增长
    - 每次恢复时，只需要扫描日志区中最近一次 checkpint 之后的 log blocks，无需扫描全部 log

代价：写两遍数据；收益：快速且安全的恢复

Journaling

- **Linux ext3 的例子**
  - **分成很多独立的 group，每个 group 都有 bitmap, inode, data block**
  - **增加一个日志区域，或者放在一个独立的磁盘上**

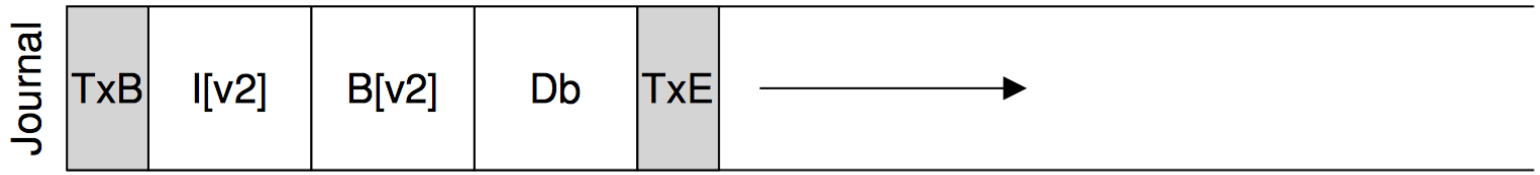
Journaling

- **Linux ext3 的例子**
  - 例如向文件中写入一个新数据块的操作包含对 **FS** 的三次写入：
    - **inode(I[v2]), bitmap(B[v2]), data block (Db)**
  - 先将这些写操作记录到日志中：

    - **TxB** 和 **TxE** 标记 **txn** 的开始和结束
    - 日志中所有的记录都会包括 **TID**，以避免不同事务的记录发生混淆

Journaling

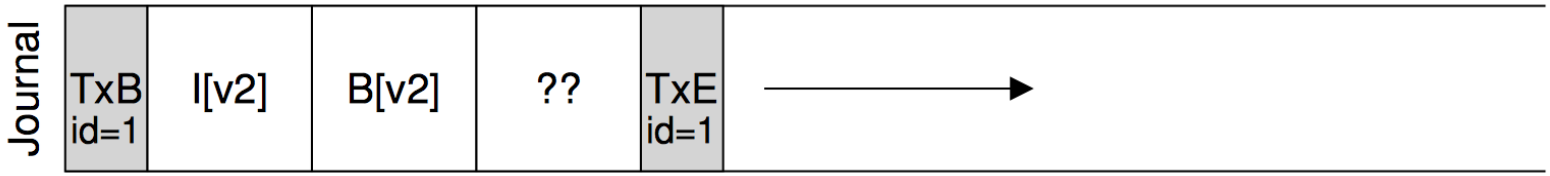
- **Linux ext3 的例子**
  - **上页的例子中， log 总共写了 5 个数据块， TxB, I[v2], B[v2], Db, TxE**



  - **写完 log 后即可向上层返回写入成功，稍后再进行 checkpoint 、真正完成对 FS 中 bitmap 、inode 和 data block 的修改**

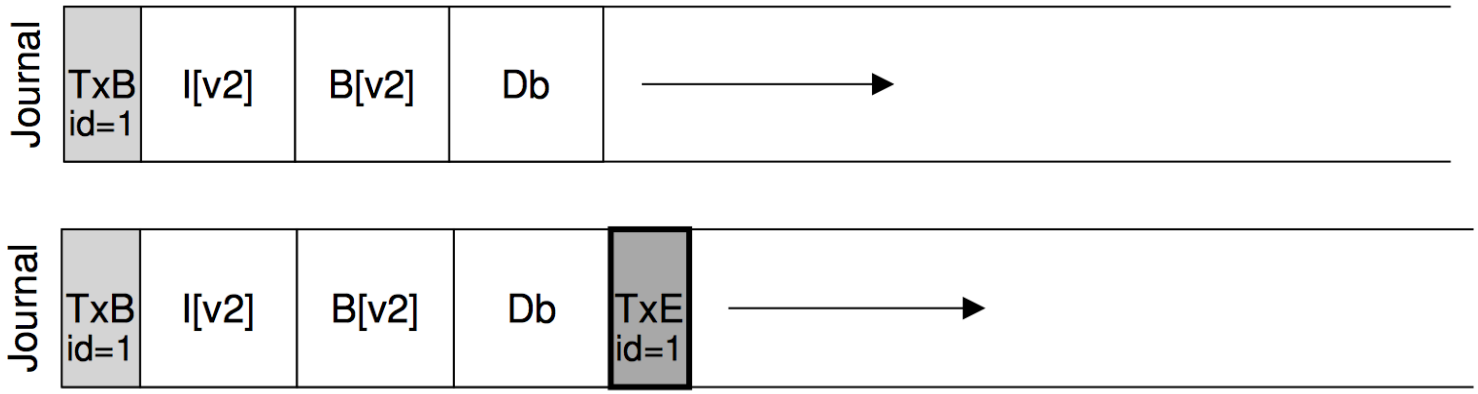
Journaling

- **5 个 log block 怎么写入?**
  - 逐个写入? 过慢
  - 一起写? 不安全
    - **block device 可能乱序写入，导致 TxE 落盘了，但是之前的 log block 还没落盘**
    - **恢复时会把 ?? 当做真正的数据内容去恢复**



Journaling

- 改进：除了 TxE 之外一起写， TxE 最后单独写



  - 存储设备保证 sector/block 的写入是原子的，要么写成功，要么完全没写；所以 TxE 要小于 512B 且 sector align

Journaling

- 总结：Journaling (WAL) 分三步
  - Journal Write （将 TxB 和对 medata 、 data 的修改写入 log ）
  - Journal Commit （将 TxE 写入 log ）
  - Checkpoint （完成对 metadata 和 data 的最终修改，清理 log ）

- 进一步改进提升性能：
  - 对 log 增加 checksum
    - 这样 TxE 可以和其他 log block 可以一起写，省掉了第二步
    - 例如：Linux ext4

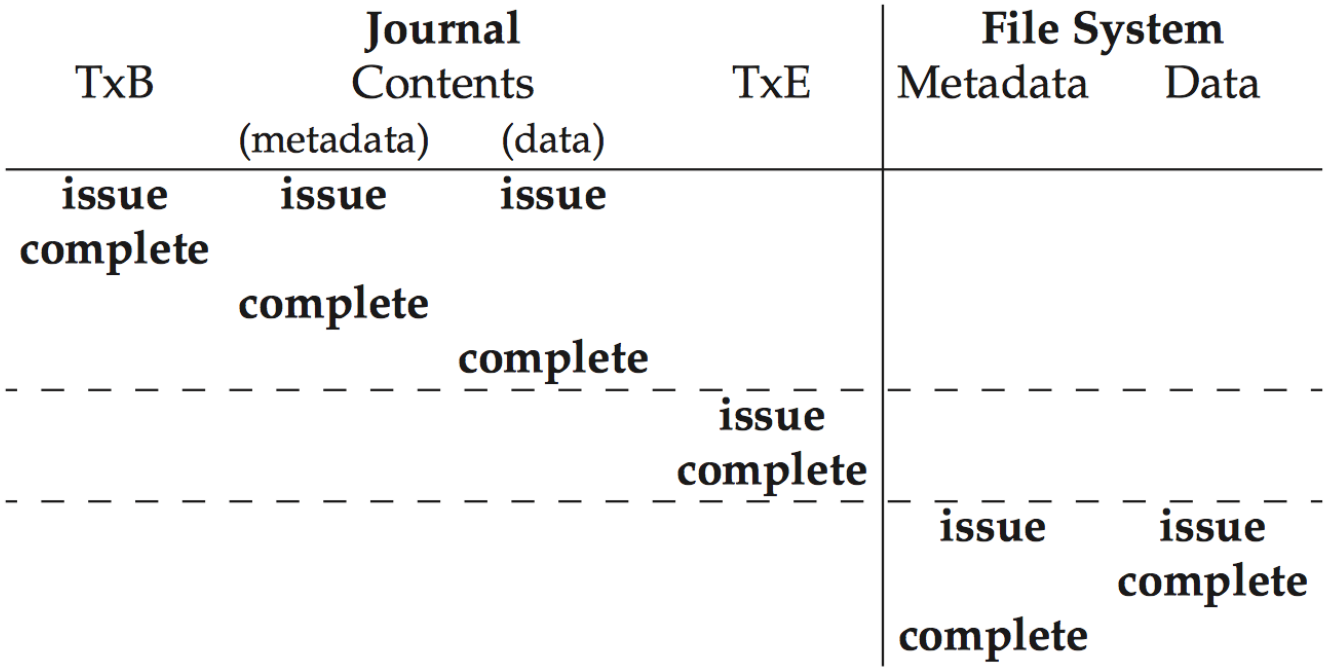
以算代存，以存代算，都是常见的优化技巧
要具体看哪个收益大、代价小

Journaling

- Recovery
  - 如果 commit 完成前 crash
    - 忽略该 transaction 对文件系统的修改
  - 如果 commit 完成后、 checkpoint 完成前 crash
    - 则从 TxB 开始 replay 这些 updates (redo log)
  - 如果 checkpoint 完成后 crash
    - No one cares:)

Journaling

- **Journaling timeline**

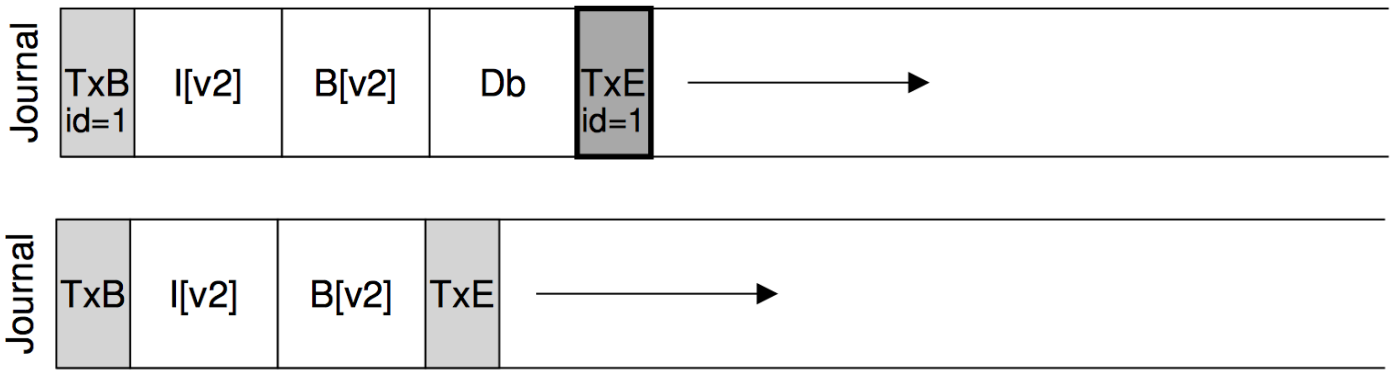


Journaling

- 如何减少写日志的磁盘开销
  - 例如写入同一个目录下的两个文件 file1 和 file2
    - 要多次更新 bitmap，父目录的 data
  - 按照上述 log protocol， I/O 次数很多
  - 如何优化?
    - batch commit (i.e., batching log updates)
    - 将近段时间内的 tx 看做一个 large tx or global tx，批量执行
    - 批量化的思想在 CS 中非常常见，本质是牺牲 latency 换取 throughput

Journaling

- **Metadata Journaling**
  - 目前方案的问题：两倍 I/O traffic
  - 解决方案：
    - 日志中去掉 data block，减少绝大多数 I/O
    - Db 直接写到原位置，避免重复写



Db 什么时候写入?

Journaling

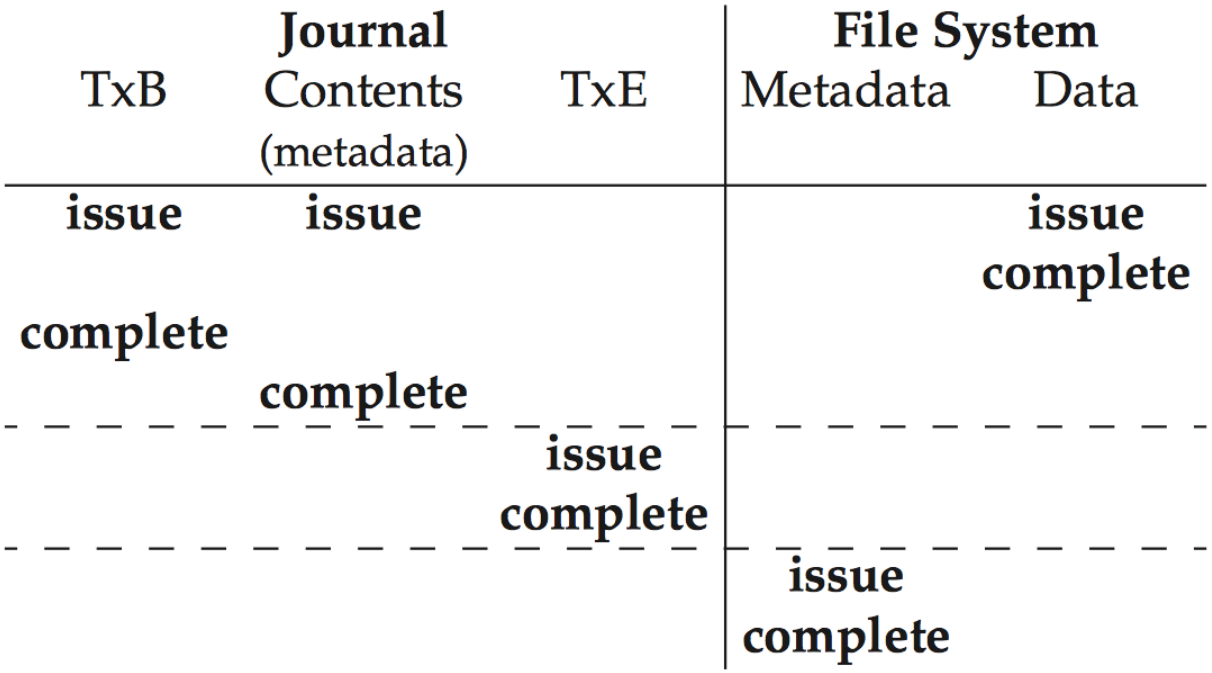
- **Metadata Journaling**
  - **Db 什么时候写入?**
    - **在 TxE 之后，不行：恢复时可能没有正确的 Db**
    - **在日志之前：**
      - **1. Data Write: in-place write to final location**
      - **2. Journal Metadata Write**
      - **3. Journal Commit, wait for 1 to complete**
      - **4. Checkpoint Metadata and free (释放日志空间)**

避免在 log block 和 data block 中两次写入用户数据
由于用户数据通常远大于 metadata ，这样可以节省大量 I/O

1 和 2 的顺序可以交换吗?

Journaling

- **Metadata journaling timeline**

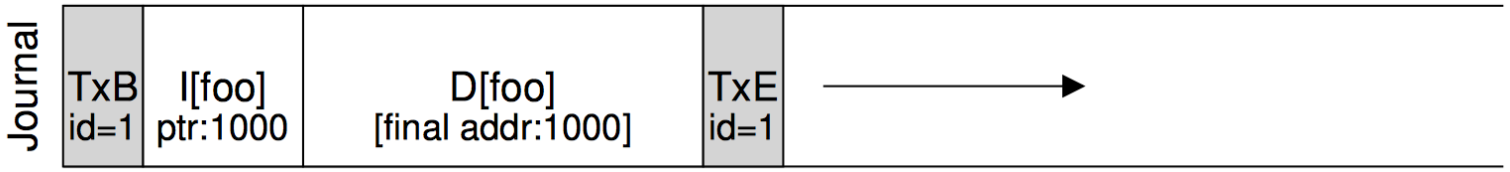


Journaling

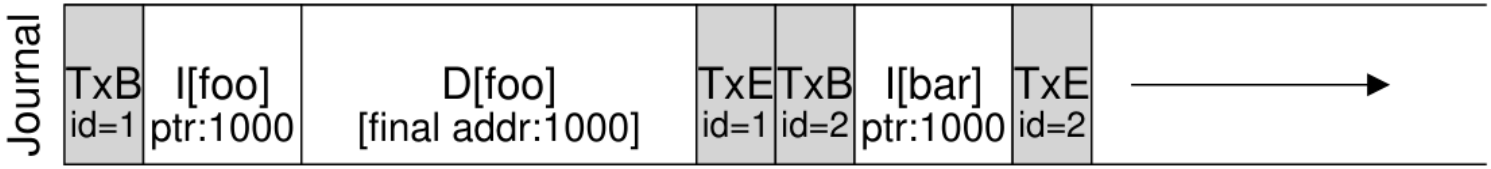
- **Metadata Journaling Applications:**
  - **Windows NTFS, SGI's XFS**
  - **Ext3 可选 ordered journaling mode , 等价于 metadata journaling**

Journaling

- **Metadata Journaling 的 bug：**
  - 假设创建一个目录 foo，其 data block 为 1000
    - 目录的 data block 被认为是元数据，所以写入日志



  - 然后删除 foo 目录，之后再写入一个新的文件 bar，刚好 reuse data block 1000



会有什么问题?

Journaling

- **Metadata Journaling 的 bug：**
  - **bar 文件的 data block 会直接写到 data block 1000，不在 log 中记录**
  - **但是当恢复时，会先把目录 foo 的 data block 内容恢复到 data block 1000，覆盖 bar 文件的内容**

这个问题的根本原因是什么?
如何解决?

Journaling

- **Metadata Journaling 的 bug：**
  - **ext3 的解决方案:**
    - **在日志中增加 revoke （删除） record**
    - **恢复时先扫描日志，如果有 revoke , revoked data （例如 foo 的 data block 1000 ）不会被恢复**

为什么不把 directory 的 data block 当成 data？

Journaling

- **Solution #3: Other Approaches**
  - **Soft Updates**
    - 精心设计 FS 的内部结构和写入顺序
      - 例如先写 data block 再写 indrect nodes，最后再写 inode 可以避免 inode 指向 dirty data blocks
    - 实现比较复杂

Journaling

- **Solution #3: Other Approaches**
  - **Copy-On-Write (COW)**
    - 用于 **Log-structured FS, ZFS**
    - 永远不原位更新，而是写到新的位置上
    - 本质上是一种多版本（**multi-versioning**）的实现

回想一下 Flash SSD 中的 hybrid mapping?

Journaling

- **Solution #3: Other Approaches**
  - **Backpointer-Based Consistency (BBC)**
    - 数据块中增加 back pointer，指向 inode
    - 通过判断 forward pointer 和 back pointer 是否一致，来判断 inode 和 data block 是否一致
      - back pointer 起到对 parent 的校验作用

是不是有点像区块链的味道?

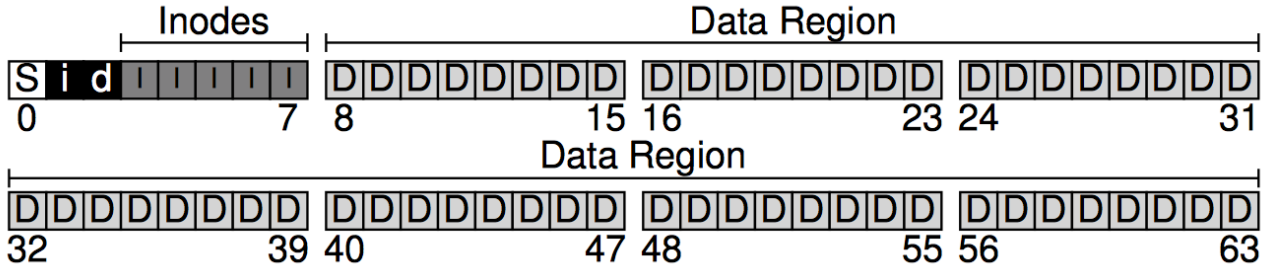
F2FS (Flash-friendly File System)

- 三星最早开发，华为、 Google 等参与开发
- 也是基于 Copy-on-write (COW) 思想
- 应用于安卓设备和部分 Linux 服务器

F2FS: A New File System for Flash Storage
Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho
https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee

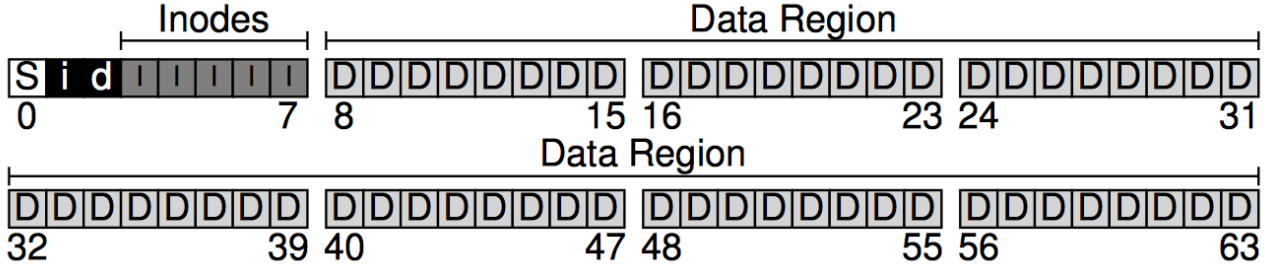
课堂练习

- 一个文件系统的数据分布类似 VSFS， inode 包括 15 个指针（12 个 direct, 2 个 indirect, 1 个 double indirect）。写入一个 10MB 的 /foo/bar 文件，分别采用 data journaling 和 metadata journaling，分别需要多少次 I/O？（除了文件 data block 外的都算元数据，每个 4KB block 的读或写都算一次 I/O ，创建或写入同一文件只需要最终保存一次 metadata，向 log 中写一个 record 也算写入一次 I/O）



课堂练习（答案）

- 一个文件系统的数据分布类似 VSFS， inode 包括 15 个指针（12 个 direct, 2 个 indirect, 1 个 double indirect），写入一个 10MB 的 /foo/bar 文件，分别采用 data journaling 和 metadata journaling，分别需要多少次 I/O？（除了文件 data block 外的都算元数据，每个 4KB block 的读或写都算一次 I/O ，创建或写入同一文件只需要最终保存一次 metadata，向 log 中写一个 record 也算写入一次 I/O）

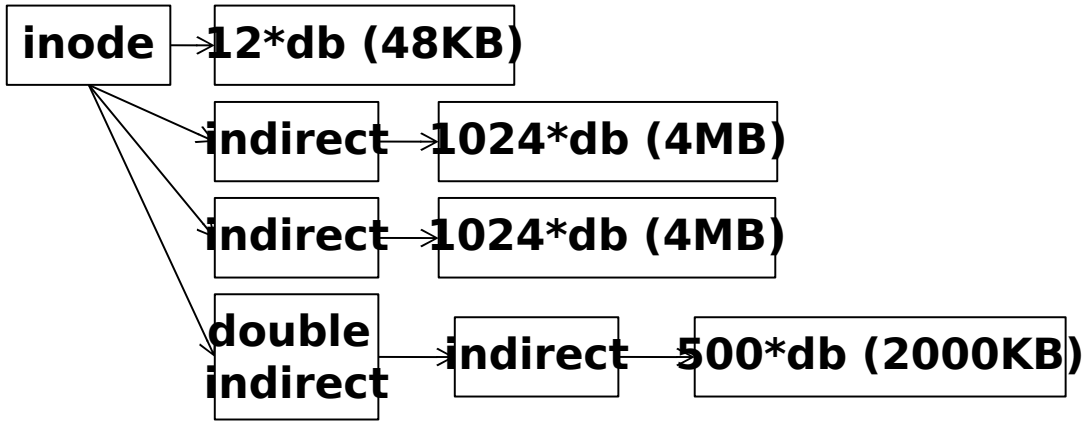


首先，读取 superblock，读取 / 的 inode 和 data block，读取 /foo 的 inode 和 data block。为了给 /foo/bar 创建 inode 和 data blocks，还需要读取 inode bitmap 和 data bitmap。所以共需要 7 次 read 来读取以上的 metadata

课堂练习（答案）

- 一个文件系统的数据分布类似 VSFS， inode 包括 15 个指针（12 个 direct, 2 个 indirect, 1 个 double indirect）写入一个 10MB 的 /foo/bar 文件，分别采用 data journaling 和 metadata journaling，分别需要多少次 I/O？（除了文件 data block 外的都算元数据，每个 4KB block 的读或写都算一次 I/O ，创建或写入同一文件只需要最终保存一次 metadata，向 log 中写一个 record 也算写入一次 I/O）

然后需要为 /foo/bar 创建 1 inode、 3 indirect pointer blocks、 1 double indirect pointer block、 2560 data blocks，这需要写入 inode bitmap、 data bitmap、 1 个 inode、 4 个 pointer blocks、 2560 个 data blocks。再加上修改 /foo 的 data block 和 inode，共需要 9 次 metadata 写入（inode bitmap、 data bitmap、 /foo/bar inode、 /foo inode、 /foo data block、 4 个 pointer blcoks）和 2560 次 data 写入



课堂练习（答案）

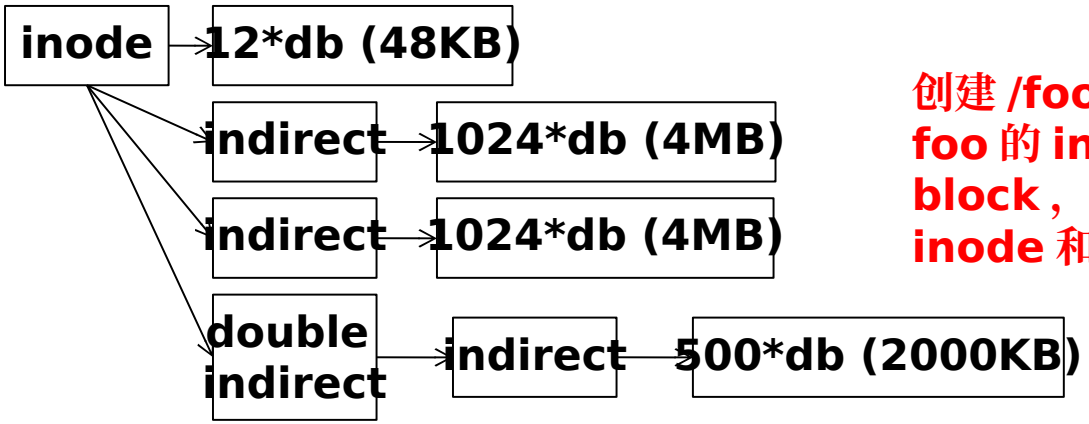
- 一个文件系统的数据分布类似 VSFS， inode 包括 15 个指针（12 个 direct, 2 个 indirect, 1 个 double indirect）写入一个 10MB 的 /foo/bar 文件，分别采用 data journaling 和 metadata journaling，分别需要多少次 I/O？（除了文件 data block 外的都算元数据，每个 4KB block 的读或写都算一次 I/O ，创建或写入同一文件只需要最终保存一次 metadata，向 log 中写多个 record 也算写入一次 I/O）

TxB: Transaction Begin; TxE: Transaction End

读取 metadata 需要 7 次 I/O

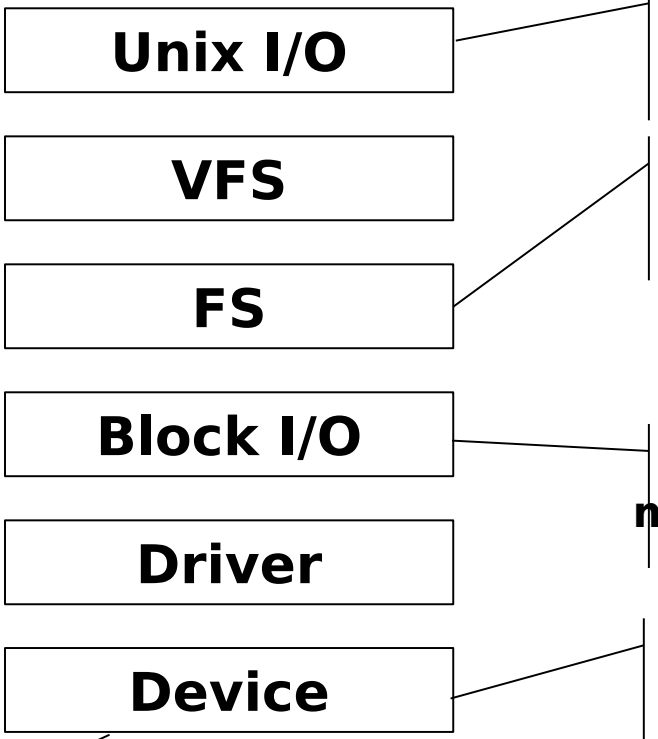
创建并写入 /foo/bar 以及修改父目录 /foo 共需要 2569 次 I/O

对于 data journaling，需要额外 2569+2 次 I/O 来写 log；对于 metadata journaling，需要额外 9+2 次 I/O 来写 log



创建 /foo/bar 只需要修改 / foo 的 inode 和 data block，不需要修改 / 的 inode 和 data block

IO-FS 复习



POSIX
Kernel data structure for files

directory tree data model
mount/umount
inode
VSFS/FFS/Ext3
consistency (fsck/journaling)

bio
md (multiple-device, software RAID)
I/O schedulers

HDD:
cylinder,head,track,sector
positioning time
scheduling (FSCAN/CSCAN)
RAID (0/1/5...)

SSD:
block erase,page write, page read
address mapping (block,page,hybrid)
garbage collection
wear-leveling

其他： **page cache...**

课堂小测：以下 AI 的说法是否正确?

TLB 是 CPU 内部高速缓存，存放最近使用过的虚拟页—物理页映射；页表存放在主存，是系统完整的地址映射表。
二者的有效位（valid bit）相互独立维护，不是同步实时更新。

地址翻译时， CPU 先查 TLB：

- 若 TLB 命中（有效），直接使用物理地址，不会再去访问页表；
- 只有 TLB 未命中，才遍历页表。

因此 TLB 有效不校验页表状态。

TLB 有效、页表无效的典型场景：

Outline

- Linux I/O Stack
- File System
- File System Implementation
- Fast File System
- FSCK and Journaling
- Log-structured File Systems （不讲）
- Data Integrity and Protection （不讲）

Log-structured File Systems

- **Log-structured File Systems**
  - **Early 90's, Berkeley, Professor John Ousterhout and graduate student Mendel Rosenblum**
- **Motivation:**
  - **System memories are growing**
    - **More data are cached in memory, especially reading**
    - **File system performance is determined by writing**
  - **There is a large gap between random I/O performance and sequential I/O performance**

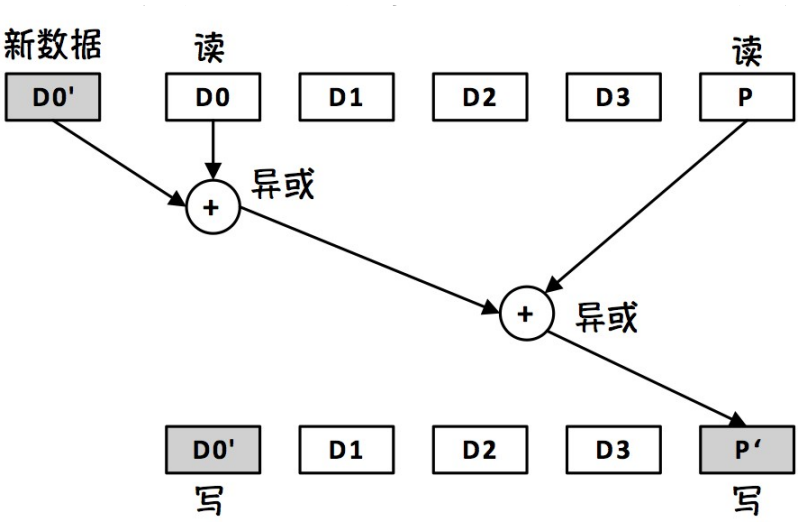
Log-structured File Systems

- **Motivation:**
  - **Existing file systems perform poorly on many common workloads**
    - **E.g.，当创建很多只有一个 block 的小文件时， FFS 会进行大量的写操作**
      - **one for a new inode,**
      - **one to update the inode bitmap,**
      - **one to the directory data block that the file is in,**
      - **one to the directory inode to update it,**
      - **one to the new data block that is a part of the new file**
      - **and one to the data bitmap to mark the data block as allocated**

当采用延迟写策略时，哪些 I/O 可以减少?

Log-structured File Systems

- **Motivation:**
  - **File systems are not RAID-aware**
    - **FS 认为下面是 disk，实际下面多数是 RAID （透明）**
    - **both RAID-4 and RAID-5 have the small-write problem where a logical write to a single block causes 4 physical I/Os to take place (两次读 / 两次写)**
    - **FFS 等** （前面的例子）

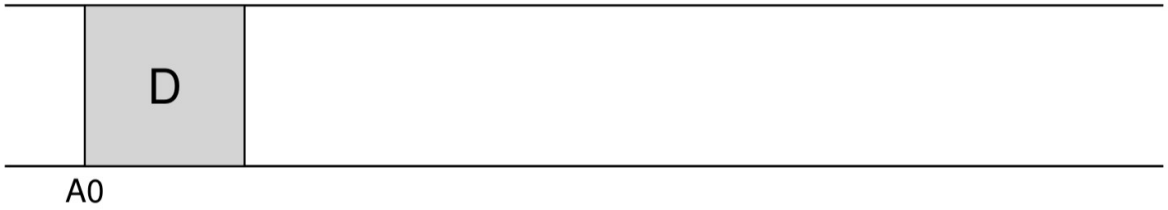


Log-structured File Systems

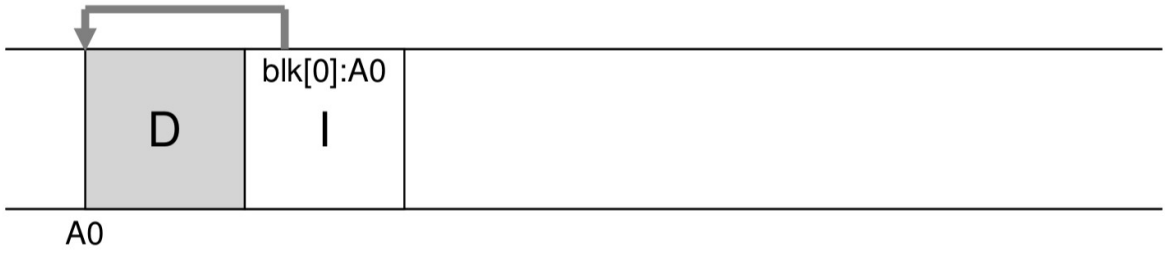
- 设计目标:
  - 优化 FS 的写性能
  - 尽量大粒度连续写，减少小粒度随机写
- **LFS Basic Idea:**
  - 对于写入操作，现在 memory segment 中 buffer（包括元数据）
  - 当 segment 满了之后，就写入外存 => 大粒度连续写
  - 不原位更新，只是在后面追加

Log-structured File Systems

- 第一个挑战：
  - 如何把随机写入变为连续写入?
  - E.g.，往文件中写一个数据块 D

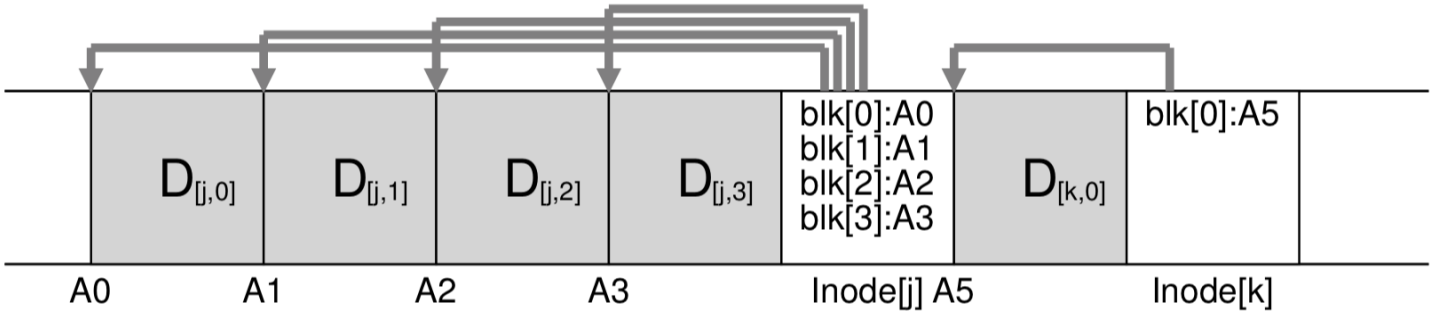


  - 但是文件的元数据（inode）也需要写入外存
    - 一般 data block 更大（例如 4KB），而 inode 可能是 128B



Log-structured File Systems

- 为了让磁盘进行连续写入， LFS 设置了 disk buffer，加大写入粒度
  - 例如以下两批更新、 7 个数据块，一起写入磁盘



Log-structured File Systems

- LFS 中内存应该缓存多少数据呢?
  - 缓存越多，写入磁盘的性能越好，但同时消耗的内存 buffer 越大，数据安全风险也更大
  - 所以关键是缓存数据量增大对写入性能提升的收益有多大?
    - 直观想象，逐渐增大，收益越来越小
    - 应该增大到一定程度为止

Log-structured File Systems

- **LFS 中内存应该缓存多少数据呢?**
  - 假设磁盘寻道 + 旋转延迟是 $T_{position}$，内部传输率是 R_{peak} MB/s
  - 一次写入数据量为 D MB，则写入 D 花费的时间为:

$$T_{write} = T_{position} + \frac{D}{R_{peak}}$$

  - 写入效率 R：（R 应该越接近 R_{peak} 越好）

$$R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}$$

  - 设二者之间的比率为 F （0<F<1），则
    - $R_{effective} = F \times R_{peak}$.

Log-structured File Systems

- **LFS 中内存应该缓存多少数据呢?**
  - 公式变形:

$$R_{effective} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak}$$

$$D = F \times R_{peak} \times (T_{position} + \frac{D}{R_{peak}})$$

$$D = (F \times R_{peak} \times T_{position}) + (F \times R_{peak} \times \frac{D}{R_{peak}})$$

$$D = \frac{F}{1-F} \times R_{peak} \times T_{position}$$

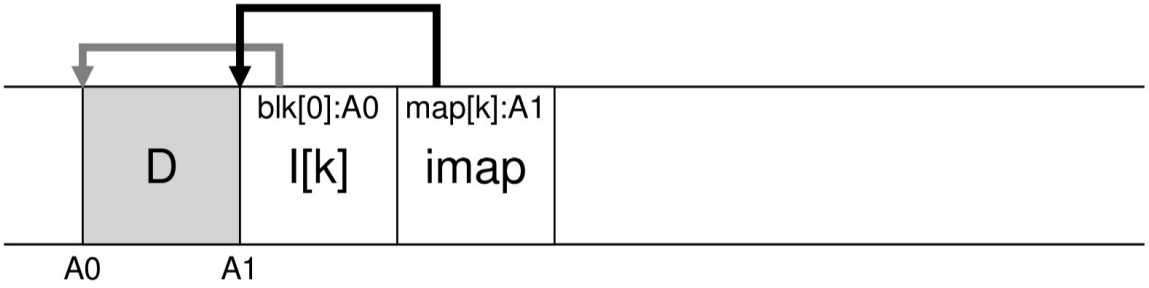
  - 所以，内存缓冲区大小 D 取决于磁盘的特性，也取决于设计的目标 F （期望达到峰值性能的多少）
    - 例如， **D=0.9/(1-0.9) × 100 MB/s × 0.01 seconds = 9 MB**

Log-structured File Systems

- 新的问题:
  - Data block 和 inode 都写入了 disk，通过 inode 可以找到 data block
  - 但是如何找到 inode 呢?
    - 传统 FS 中， inode 放在固定区域，通过上一级目录的 data block 可以找到下一级文件 / 目录的 inode

Log-structured File Systems

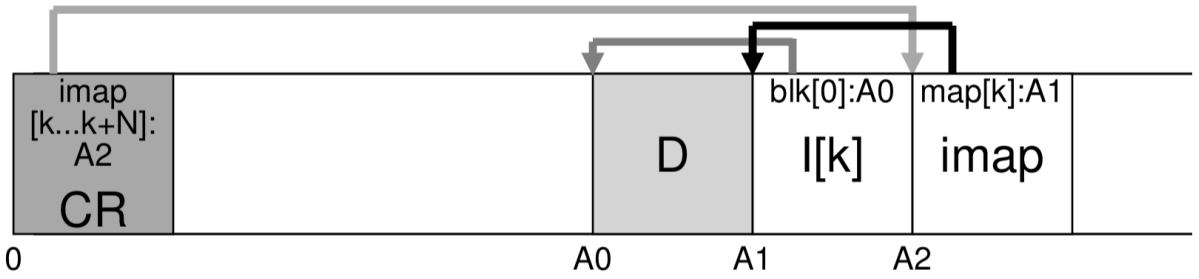
- LFS 做法:
  - 引入 inode map (imap)，根据 inode 编号，找到 inode 所在的磁盘地址
  - 在 inode 更新，并写入 disk 时，也追加写入新版本的 inode map



  - 下一个问题:
    - 如何找到这个新的 imap？

Log-structured File Systems

- 如何找到新的 imap？
  - 可以一级一级把更新后的指针写入 disk （异位更新, copy-on-write）
  - 但是最终磁盘中需要有原位更新的地方（ root ）
    - check-point region (CR)：指向所有 imap
    - CR 在 LFS 中周期性更新， 30s 一次，对性能影响不大

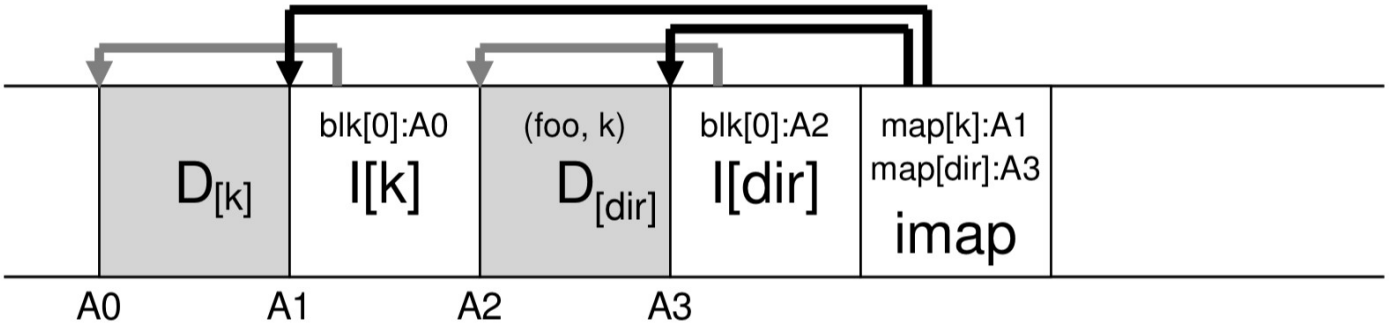


Log-structured File Systems

- 从 LFS 中读取数据
  - check-point region
  - Entire imap, cache it in memory
  - 根据 inode number，从 imap 中找到 inode 地址
  - 从磁盘中读取 inode，找到目标 data block 的地址
  - 读取 data block

Log-structured File Systems

- 如何处理目录?
  - 目录的 **data block** 中包含子文件 / 子目录的 **inode number**
  - **Imap->A3(I[dir])->A2(D[dir]) ->** 目标文件的 **inode number k**
  - **Inode address = imap(k) -> A1(I[k]) -> A0(D[k])**

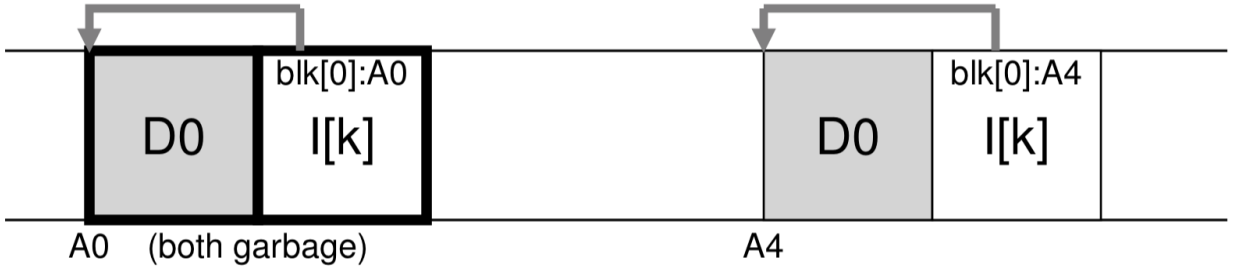


Log-structured File Systems

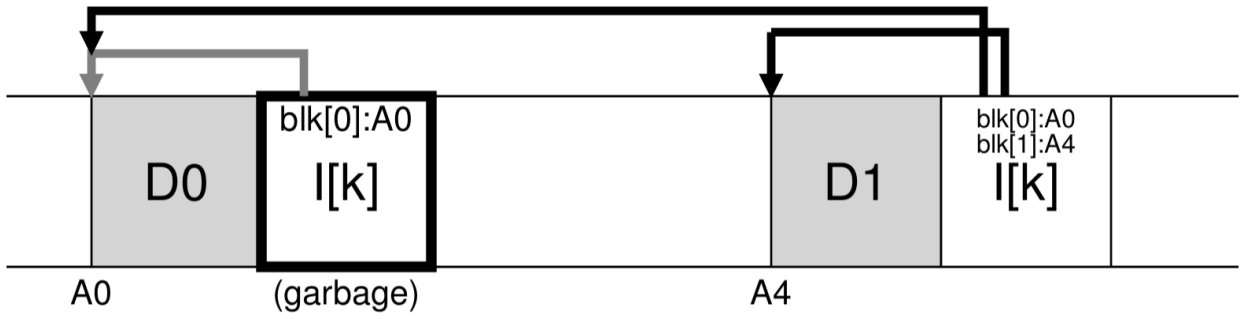
- **recursive update problem**
  - **LFS 中除了 CR，都是异位更新**
  - **因此更新一个文件，以上每一级目录的 data block 和 inode 都要对应更新**
  - **因此写放大比较严重，性能也有较大挑战**

Log-structured File Systems

- **Garbage Collection**
  - 旧版本数据要释放，回收空间
  - 如果 inode 和 data block 都成为垃圾，则一起回收



  - 也可能只是在文件后追加了一个新块，只有 inode 修改



Log-structured File Systems

- **Garbage Collection**
  - 实际中采用 segment-by-segment 的方式进行清理
  - 可能会搬移部分数据来清理 segment (compaction)
  - 选择待清理 segment 的策略:
    - 有很多研究论文
    - 原版 LFS 论文的方法：分为 hot segment 和 cold segment
      - Hot: 频繁 overwritten，尽量延迟清理，这样里面的数据最后可能都变为垃圾
      - Cold：只要少部分数据 overwritten，多数数据 stable，所以应该尽快清理

Log-structured File Systems

- 借鉴 LFS 思想的文件系统
  - NetApp's WAFL
  - Sun's ZFS
  - Linux btrfs [R+13]
  - modern flash-based SSDs (Flash 内部本身就不支持原位更新，比较契合)
  - adopt a similar copy-on-write approach to writing to disk

Outline

- Linux I/O Stack
- File System
- File System Implementation
- Fast File System
- FSCK and Journaling
- Log-structured File Systems
- Data Integrity and Protection

Data Integrity and Protection

- 目标：保证数据安全
  - Ref: 之前的 RAID，还有纠删码（Erasure Code)
  - Data Integrity（数据完整性）
  - Or Data Protection
- 提高数据安全性的共同思路
  - 更可靠的硬件
    - 数据长期保存（石头，竹简，纸张，电子器件， DNA）
  - 额外存储 / 分散存储
    - 同时出问题的概率大大降低了

Data Integrity and Protection

[例 1] 假设单个磁盘的 MTTF 为 1000000 小时，MTTR 为 8 小时，两个磁盘构成的 RAID1 系统的 MTTF 为多少?

MTTF： Mean Time To Failure，连续正常工作的时间

一般 1/MTTF 可以认为单位时间（小时）的出故障概率

MTTR： Mean Time To Repair，平均修复时间

MTBF： Mean Time Between Failure，两次失效之间的时间差，等于 MTTF+MTBF

Data Integrity and Protection

- 单个磁盘的出故障概率 =1/MTTF
- RAID 1 中两个磁盘同时发生故障的概率为

  2/MTTF*(MTTR*(1/MTTF)) = $2*MTTR/MTTF^2$
- 所以， RAID 1 系统的 MTTF = $MTTF^2/(2*MTTR)$

  = $1000000^2/16$ （约为 10^{11} ）
- 冗余可以显著提升数据安全性

课堂练习

- 单个磁盘的 MTTF 和 MTTR 已知，那么一个 5 个磁盘构成的 RAID 5 磁盘系统的 MTTF 为多少？（可以近似求解）

练习答案

- 单个磁盘的 MTTF 和 MTTR 已知，那么一个 5 个磁盘构成的 RAID 5 磁盘系统的 MTTF 为多少?
  - RAID 5 能够容忍单盘错，但是无法容忍双盘或双盘以上错误，即两个磁盘故障，则是系统故障
  - 5 个盘中任意一个出故障的概率为 5/MTTF
  - 任意一个盘出错后，在维修好之前第二个盘坏的概率为 5/MTTF * (MTTR*4/MTTF)

    $=20*MTTR/MTTF^2$
  - 所以 RAID 5 的 MTTF 为 $MTTF^2/(20*MTTR)$，远大于单盘的 MTTF

Data Integrity and Protection

- 单个磁盘内部
  - 可能个别 bits 损坏（例如因为 head 划伤表面）
- Flash
  - 某些 block 可能因为磨损过多而提高损坏概率
  - 注意：磨损不是导致确定性损坏，而是提高概率
- 解决思路：冗余存储（不太会大面积划伤
  - 问题：多备份开销太大，磁盘有效空间成倍缩小
  - 方法：部分冗余（类似 RAID ）
    - error correcting codes (ECC)
      - 异或运算

Data Integrity and Protection

- 磁盘故障
  - **Latent sector errors**
    - **Sector** 无法访问 / 访问出错
    - 上层知道出问题了
  - **Block corruption**
    - **Block** 能够访问，当上面的数据不对了
    - 如果不做进一步数据检测，上层不知道出问题了

	Cheap	Costly
LSEs	9.40%	1.40%
Corruption	0.50%	0.05%

Data Integrity and Protection

- 磁盘故障
  - **Latent sector errors**
    - 通过 ECC 等冗余存储可以恢复原始数据
  - **Block corruption**
    - 首先要能够检测读出等数据是否正确?
    - 也需要冗余存储才能判定（no such thing as a free lunch）
    - 但是不用像 ECC 等达到能恢复数据的程度，只需要检测
    - Checksum （校验码）
    - E.g.，身份证最后一位

Data Integrity and Protection

- **Common Checksum Functions**
  - 假设 **block** 为 **16B**（实际 **sector** 要大很多），**checksum** 大小为 **4B**
  - **16B** 原始数据（**16** 进制）：
    - **365e c4cd ba14 8a92 ecef 2c3a 40be f666**
  - 转为二进制
    - **0011 0110 0101 1110 1011 1010 0001 0100**
    - **1110 1100 1110 1111 0100 0000 1011 1110**
    - **1100 0100 1100 1101 1000 1010 1001 0010**
    - **0010 1100 0011 1010 1111 0110 0110 0110**

Data Integrity and Protection

- **Common Checksum Functions**
  - 每 4B 数据一行，每列进行 XOR 运算
    - 0011 0110 0101 1110 1011 1010 0001
    - 1110 1100 1110 1111 0100 0001 1010
    - 1100 0100 1100 1101 1000 1010 0011
    - 0010 1100 0011 1010 1111 0110 0110
  - ----------
    - 0010 0000 0001 1011 1001 0100 0011
    - 0x201b9403

Data Integrity and Protection

- **Common Checksum Functions**
  - 每次读取数据时，重新计算一次 checksum
  - 然后和存储的 checksum 比对，完全一致说明数据没问题
    - 出错时也可能是 checksum 本身错了（block corruption），而原始数据没错，但是这种方法无法分辨，也会报错
  - XOR 操作简单，但是每一列只能容忍单 bit 错误（翻转）
  - 如果每列有 2 个 bit 错误，则 checksum 会认为没错

Data Integrity and Protection

- **Fletcher checksum**
  - **John G. Fletcher [F82]**
  - **A block D consists of bytes d1 ... dn**
  - **Compute two check bytes, s1 and s2**
  - **s1 = (s1 + di) mod 255 (computed over all di);**
  - **s2 = (s2 + s1) mod 255 (again over all di)**
  - **almost as strong as the CRC**
  - **detecting all single-bit, double-bit errors, and many burst errors**

Data Integrity and Protection

- **Cyclic Redundancy Check (CRC)**
  - **treat D as if it is a large binary number (it is just a string of bits after all)**
  - **and divide it by an agreed upon value (k)**
  - **The remainder of this division is the value of the CRC**
  - **CRC 在网络领域应用也很多**

Data Integrity and Protection

- **Checksum Layout**
  - 原始数据:

  - 增加 checksum
    - 例如每个 sector (512B) 增加 8B 的 checksum
    - 一种实现方法是磁盘制造时就每个 sector 520B，用户可见 512B

Data Integrity and Protection

- 如果磁盘不是这样，那 Fs 需要安排 checksum 的存储位置，可能是这样：
  - 一组 sector 的 checksum 集中放在一个 sector 中

Data Integrity and Protection

- **Scrubbing**
  - 用户访问数据时可以检测数据是否正确
  - 但多数数据很少访问，所以需要有后台操作，周期性检测一遍所有数据，即 **disk scrubbing** （擦洗）
  - 读出数据，计算 **checksum** ，比较
  - 如果不匹配，尝试从冗余数据（ **ECC** ）中恢复

文件操作对修改时间的影响

- **https://blog.zenggyu.com/zh/post/2022-06-04/%E5%B8%B8%E8%A7%81%E6%93%8D%E4%BD%9C%E5%AF%B9%E6%9

对象类型	操作类型	文件系统种类/时间戳类型							
		EXT4				NTFS			
		创建时间	访问时间	更改时间	改动时间	创建时间	访问时间	更改时间	改动时间
文件	创建文件	✓	✓	✓	✓	-	✓	✓	✓
	访问文件	x	✓	x	x	-	✓	x	x
	更改文件	x	x	✓	✓	-	x	✓	✓
	变更文件名称	x	x	x	✓	-	x	x	✓
	变更文件权限	x	x	x	✓	-	x	x	x
目录	创建目录	✓	✓	✓	✓	-	✓	✓	✓
	改动目录名称	x	x	x	✓	-	x	x	✓
	改动目录权限	x	x	x	✓	-	x	x	x
	在目录内创建文件	x	x	✓	✓	-	x	✓	✓
	访问目录内的文件	x	x	x	x	-	x	x	x
	更改目录内的文件	x	x	x	x	-	x	x	x
	变更目录内的文件名称	x	x	✓	✓	-	x	✓	✓
	变更目录内的文件权限	x	x	x	x	-	x	x	x
	删除目录内的文件	x	x	✓	✓	-	x	✓	✓

备注：符号"✓"表示时间戳发生了变化，符号"x"表示时间戳未发生变化，符号"-"表示时间戳不存在。