第十八讲：文件系统实例

第 1 节：FAT 文件系统

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

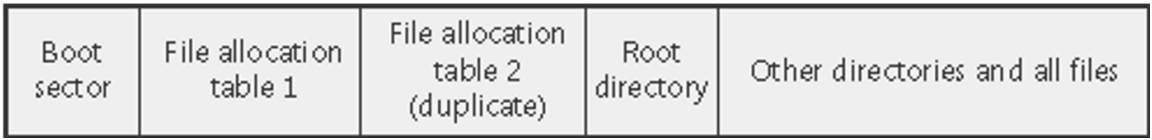
2020 年 5 月 5 日

提纲

File Allocation Table (FAT) Volume

- A simple file system originally designed for small disks and simple folder structures.
- The FAT file system is named for its method of organization, the file allocation table, which resides at the beginning of the volume.
- To protect the volume, two copies of the table are kept, in case one becomes damaged.
- The file allocation tables and the root folder must be stored in a fixed location so that the files needed to start the system can be correctly located.

Structure of a FAT Volume



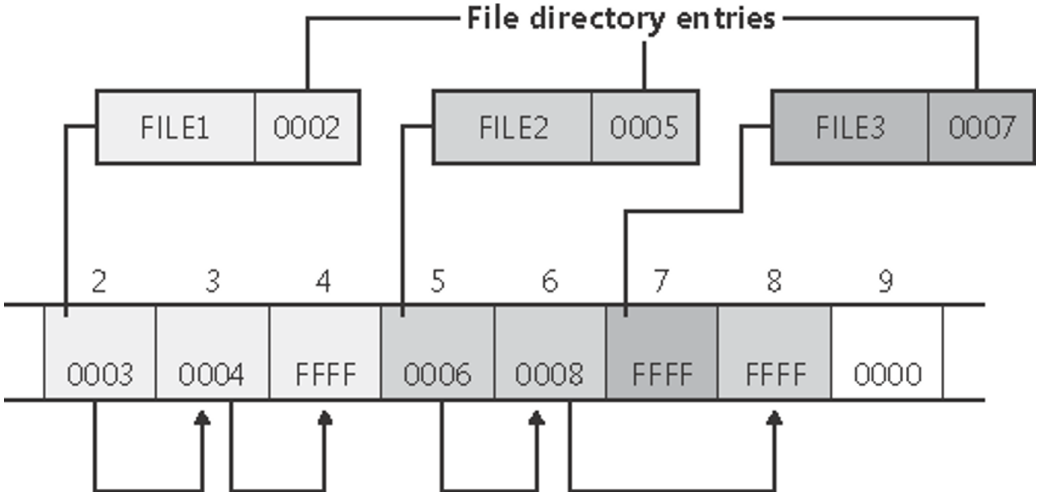
- Boot sector
- FAT1
- FAT2
- Root directory
- Other directories and all files

Differences Between FAT Systems

System	Bytes Per Cluster Within File Allocation Table	Cluster limit
FAT12	1.5	Fewer than 4087 clusters.
FAT16	2	Between 4087 and 65526 clusters, inclusive.
FAT32	4	Between 65526 and 268,435,456 clusters, inclusive.

- FAT32 is a derivative of the File Allocation Table (FAT) file system that supports drives with over 2GB of storage.
- FAT32 drives can contain more than 65,526 clusters and results in more efficient space allocation on the FAT32 drive.

Example of File Allocation Table



File Allocation System

The file allocation table contains the following types of information about each cluster on the volume (see example below for FAT16):

- Unused (0x0000)
- Cluster in use by a file
- Bad cluster (0xFFF7)
- Last cluster in a file (0xFFF8-0xFFFF)

FAT Root Folder

The root folder contains an entry for each file and folder on the root. The only difference between the root folder and other folders is that the root folder is on a specified location on the disk and has a fixed size (512 entries for a hard disk, number of entries on a floppy disk depends on the size of the disk).

Folder Entry

The Folder Entry includes the following information:

- Name (eight-plus-three characters)
- Starting cluster number in the file allocation table (16 bits)
- File size (32 bits)
- Attribute byte (8 bits worth of information)
- Create time (24 bits)
- Create date (16 bits)
- Last access date (16 bits)
- Last modified time (16 bits)
- Last modified date (16 bits)

Long Filenames on FAT Volumes

- FAT creates an eight-plus-three name for the file. In addition to this conventional entry.
- FAT creates one or more secondary folder entries for the file, one for each 13 characters in the long filename. Each of these secondary folder entries stores a corresponding part of the long filename in Unicode.
- FAT sets the volume, read-only, system, and hidden file attribute bits of the secondary folder entry to mark it as part of a long filename.

Folder Entries for the long filename

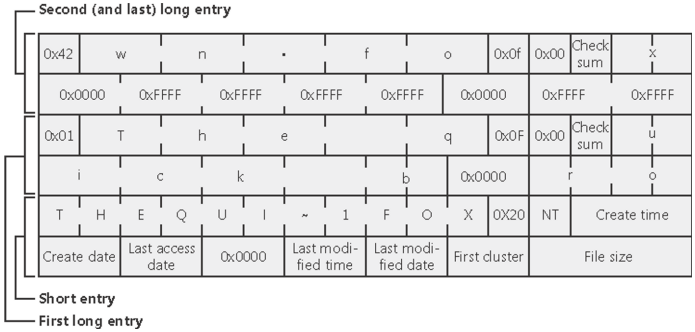


Figure shows all of the folder entries for the file Thequi~1.fox, which has a long name of The quick brown.fox. The long name is in Unicode, so each character in the name uses two bytes in the folder entry. The attribute field for the long name entries has the value 0x0F. The attribute field for the short name is 0x20.

第十八讲：文件系统实例

第 2 节：EXT4 文件系统

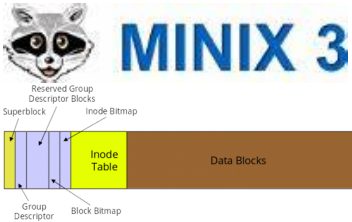
向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

历史



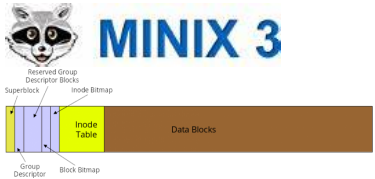
把

Linux 文件系统历史

尽管 EXT[1-4] 文件系统家族是为 Linux 编写的，但它的根源是 Minix 操作系统和 Minix 文件系统，它们早于 Linux 大约五年，于 1987 年首次发布。如果我们回顾一下 EXT 文件系统家族历史和技术从其 Minix 根源的演变，了解 EXT4 文件系统要容易得多。

Reference:
OSTEP txtbook: Crash Consistency: FSCK and Journaling; Ext4 slides from Mingming Cao; An introduction to Linux's EXT4 filesystem by David Both

历史

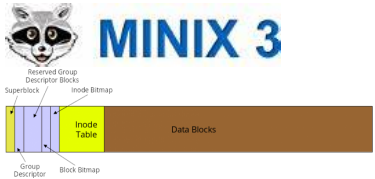


- Linus Torvalds 刚开始编写 Linux 内核时，他需要一个文件系统，但是不想编写一个文件系统。
- 拿来主义：直接采用 Andrew S. Tanenbaum 编写 Minix 文件系统

MINIX FS

- 引导扇区：在其上安装了硬盘的第一个扇区。引导块包括一个很小的引导记录和一个分区表。
- 超级块：每个分区中的第一个块是一个超级块，其中包含元数据，该元数据定义了其他文件系统结构，并将它们定位在分配给该分区的物理磁盘上。

历史



- Linus Torvalds 在编写原始的 Linux 内核时，他需要一个文件系统，但是不想编写一个文件系统。
- 拿来主义：直接采用 Andrew S. Tanenbaum 编写 Minix 文件系统

MINIX FS

- 索引节点区位图：表明哪些索引节点已使用和哪些索引节点空闲。
- 索引节点区：它在磁盘上有自己的空间。每个 inode 包含有关一个文件的信息，包括数据块的位置，即属于该文件的区域。
- 数据区位图：来跟踪使用和免费的数据区。
- 数据区：保存实际存储数据。

历史

Ext 2 3 4

- 最初的 EXT 文件系统（扩展）由 RémyCard 编写，并于 1992 年随 Linux 发行，以克服 Minix 文件系统的某些大小限制。

EXT FS

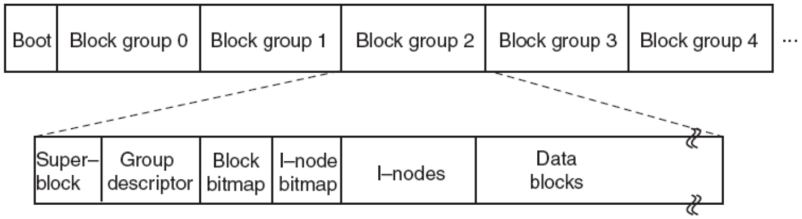
- 主要的结构更改是基于 Unix 文件系统（UFS）（也称为 Berkeley 快速文件系统（FFS））的文件系统的元数据。很少有关于 EXT 文件系统的公开信息可以验证，显然是因为它存在重大问题，并很快被 EXT2 文件系统所取代。

历史

Ext 2 3 4

- EXT2 文件系统具有与 EXT 文件系统基本相同的元数据结构，稳定可靠性有很大提升，被广泛使用于 Linux 的早期版本。

EXT2 FS

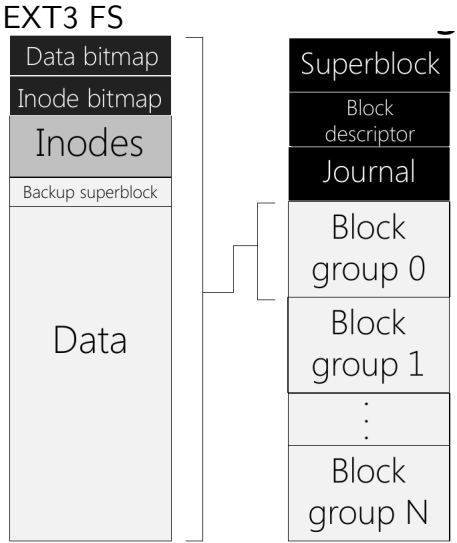


- 在一个目录下的文件存储在同一个 block group 中
- 位于不同目录下的文件分布在不同的 block group 中
- 在突然掉电或机器死机后，文件系统恢复正常要花费很长时间

历史

Ext 2 3 4

- EXT3 的唯一目标是克服花费大量时间恢复异常文件系统的时间。EXT3 文件系统增加了 journal 机制，它预先记录了将对文件系统执行的更改。其余磁盘结构与 EXT2 中的相同。



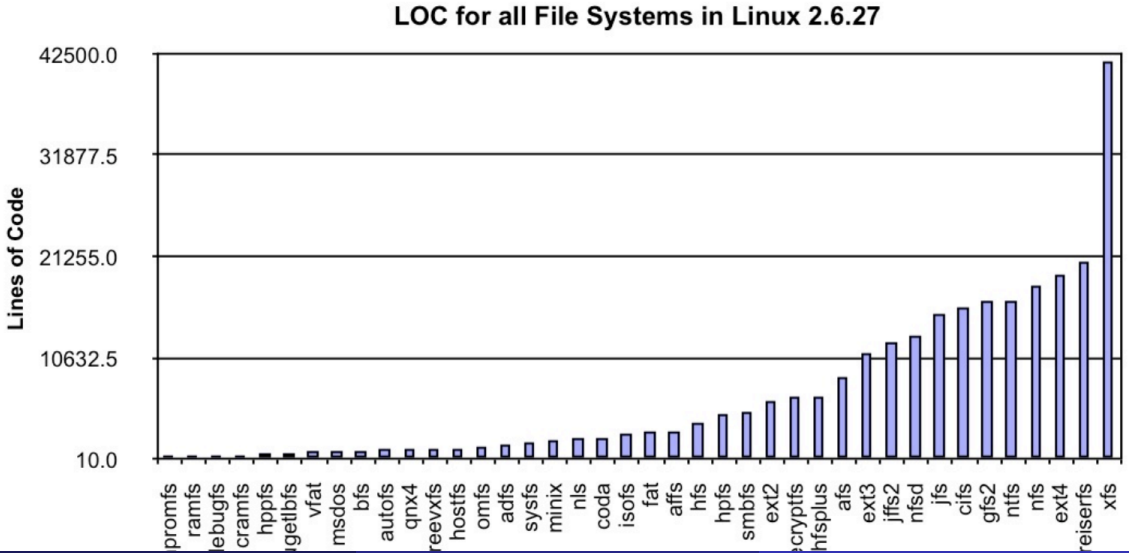
历史

- EXT4 文件系统主要进一步改善了性能，可靠性和大容量支持

EXT4 FS

- 为了提高可靠性，添加了元数据和日志校验和。
- 了满足各种关键任务要求，文件系统时间戳得到了改进，增加了纳秒精度时间间隔。
- 为提高容量和访问大容量文件的性能，把数据分配从固定块更改为扩展区

历史


LOC for all File Systems in Linux 2.6.27
Lines of Code
42500.0
31877.5
21255.0
10632.5
10.0
promfs
ramfs
debugfs
cramfs
hppfs
getlbfs
vfat
msdos
bfs
autofs
qnx4
reevxfs
hostfs
omfs
adfs
sysfs
minix
nls
coda
isofs
fat
affs
hfs
hpfs
smbfs
ext2
cryptfs
hfsplus
afs
ext3
jffs2
nfsd
jfs
cifs
gfs2
ntfs
nfs
ext4
reiserfs
xfs

细节 – 支持大容量存储

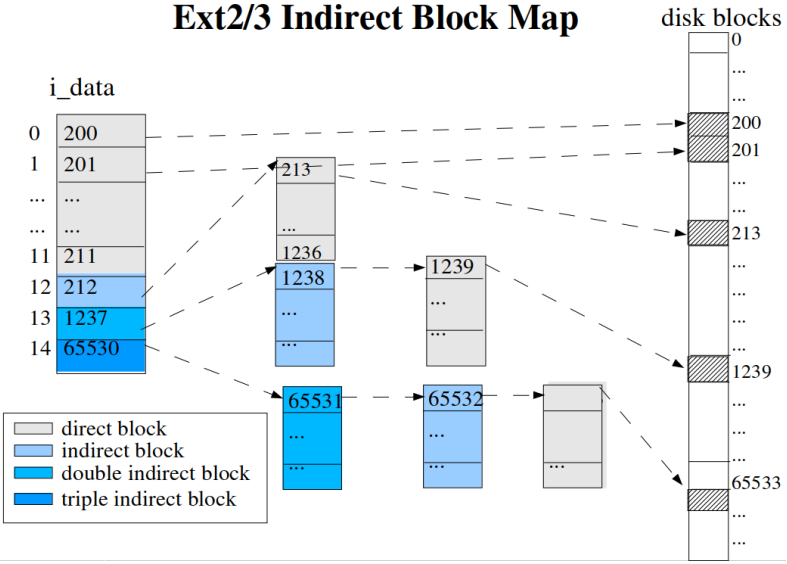
Ext4 File System

- EXT4 的重要特征：支持大容量存储和快速恢复异常状态

EXT4 FS：支持大容量存储

- 一部蓝光 8K 3D 电影，512GB～1TB
- 1EB 文件系统大小，16TB 文件大小，子目录个数无限制

细节 – 支持大容量存储


Ext2/3 Indirect Block Map
disk blocks
i_data
0
1
...
...
11
12
13
14
200
201
...
...
211
212
1237
65530
213
...
1236
1238
...
...
1239
...
...
65531
...
...
65532
...
...
...
0
...
...
200
201
...
...
213
...
...
...
...
1239
...
...
...
65533
...
...
direct block
indirect block
double indirect block
triple indirect block

细节 – 支持大容量存储

extent: 一段连续的储存块

logical	length	physical
0	1000	200

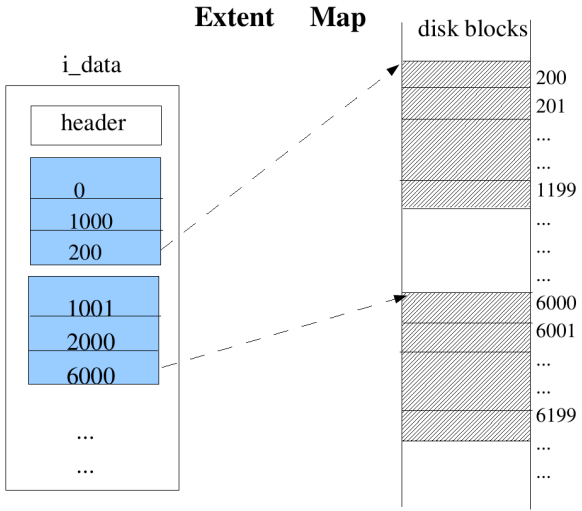
```
struct ext4_extent {
        __le32  ee_block;       /* first logical block extent covers */
        __le16  ee_len;         /* number of blocks covered by extent */
        __le16  ee_start_hi;    /* high 16 bits of physical block */
        __le32  ee_start;       /* low 32 bits of physical block */
};
```

细节 – 支持大容量存储

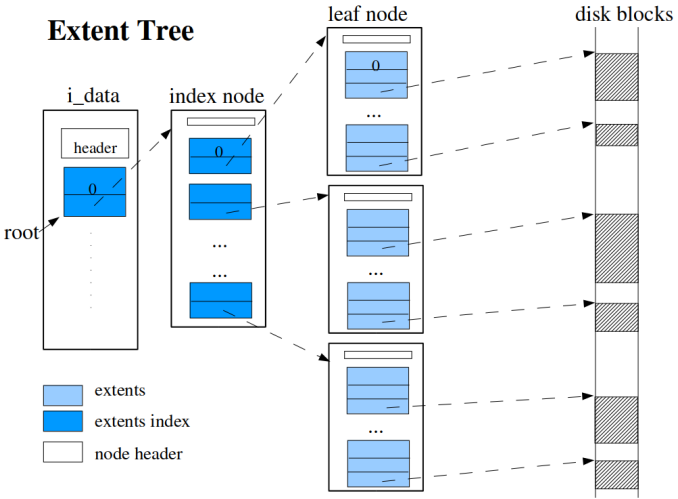
extent: 一段连续的储存块

logical	length	physical
0	1000	200

```
struct ext4_extent {
    __le32  ee_block;     /* first logical block extent covers */
    __le16  ee_len;       /* number of blocks covered by extent */
    __le16  ee_start_hi;  /* high 16 bits of physical block */
    __le32  ee_start;     /* low 32 bits of physical block */
};
```

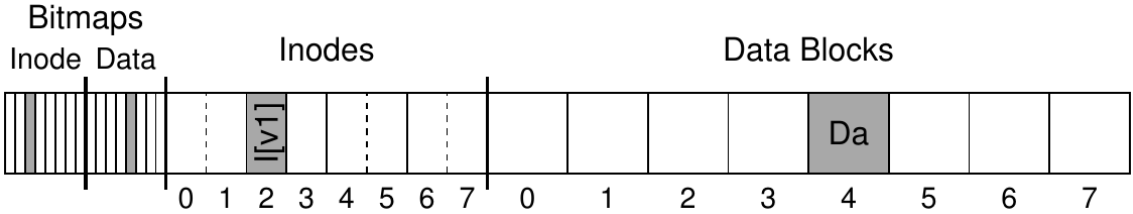


细节 – 支持大容量存储


Extent Tree
leaf node
disk blocks
i_data
index node
header
root
extents
extents index
node header

细节 – 支持恢复异常

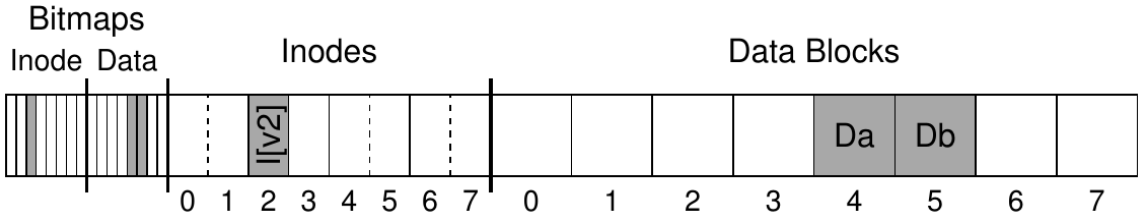
EXT4: 恢复异常文件系统 for crash-consistency problem




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

细节 – 支持恢复异常

EXT4: 恢复异常文件系统 for crash-consistency problem



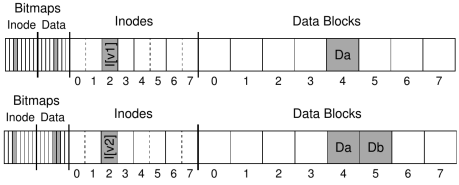

```
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

3 个写操作

- inode (I[v2])
- bitmap (B[v2])
- data block (Db)

细节 – 支持恢复异常 – Crash 场景

EXT4: 恢复异常文件系统 for crash-consistency problem



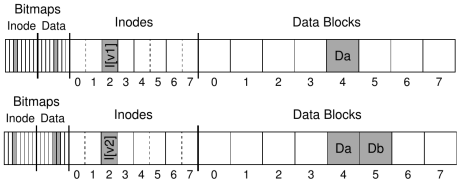

```
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

Crash 场景

- 只有 data block (Db) 写入磁盘
- 只有 inode (I[v2]) 写入磁盘
- 只有 bitmap (B[v2]) 写入磁盘
- inode(I[v2]) 和 bitmap (B[v2]) 写入磁盘
- inode(I[v2]) 和 data block(Db) 写入磁盘
- bitmap(B[v2]) 和 data block(Db) 写入磁盘

细节 – 支持恢复异常 – FSCK

EXT4: 恢复异常文件系统 for crash-consistency problem




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

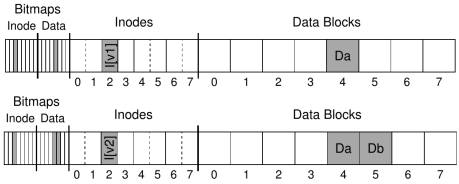
解决方法 1：File System Checker (fsck)

- Superblock: 文件系统大小大于已分配的块数
- Free blocks: bitmap v.s. Free blocks
- Inode state/links: 有效的类型字段/链接计数
- Duplicates: 两个 inode 指向同一 data
- Bad blocks: 指针显然指向超出其有效范围
- Directory：“.” 和 “..” 是头两项

太慢

细节 – 支持恢复异常 – Journaling

EXT4: 恢复异常文件系统 for crash-consistency problem




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

解决方法 2： 日志 Journaling （Write-Ahead Logging)

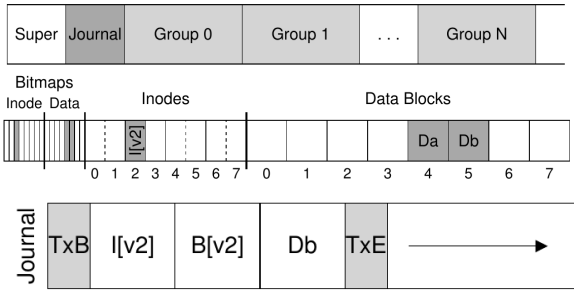
- 从数据库管理系统的世界中窃取一个想法
- 最早（1987）在 Cedar 文件系统中出现
- 出现在 EXT3/4，JFS，XFS，NTFS 等

基本思路

- 更新磁盘时，在覆盖相关结构之前，先写下一点日志（在磁盘上某个设定好的其他位置），以描述要执行的操作。

细节 – 支持恢复异常 – Data Journaling

EXT4: 恢复异常文件系统 for crash-consistency problem



方法 2： Data Journaling

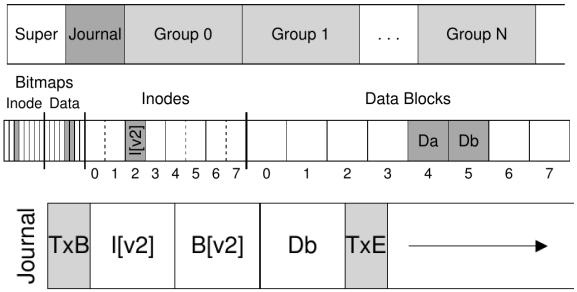
- TxB:transaction 开始
- TxE:transaction 结束
- logical logging: 中间 3 块数据

上述 transaction 写到磁盘上后, 更新磁盘，覆盖相关结构 (checkpoint)

- I[V2]
- B[v2]
- Db

细节 – 支持恢复异常 – Data Journaling

EXT4: 恢复异常文件系统 for crash-consistency problem



方法 2： Data Journaling

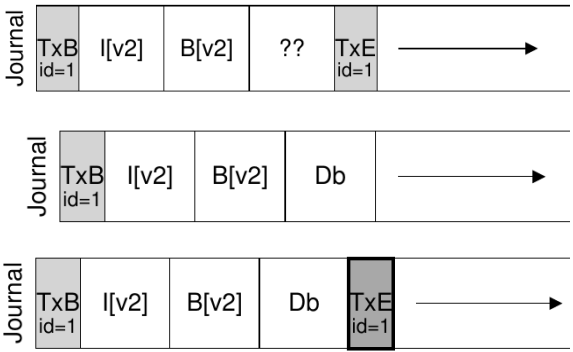
- TxB:transaction 开始
- TxE:transaction 结束
- logical logging: 中间 3 块数据

上述 transaction 写到磁盘上后, 更新磁盘，覆盖相关结构 (checkpoint)

- I[V2]
- B[v2]
- Db

细节 – 支持恢复异常 – Data Journaling

EXT4: 恢复异常文件系统 for crash-consistency problem



方法 2： Data Journaling

- Journal write
- Journal commit
- Checkpoint

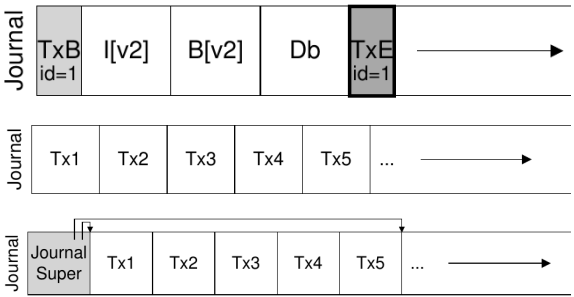
恢复 (Recovery): 在此更新序列期间的任何时间都可能发生崩溃。

- 如果崩溃发生在将事务安全地写入日志之前
- 如果崩溃是在事务提交到日志之后但在检查点完成之前发生

太多写，慢!

细节 – 支持恢复异常 – Data Journaling

EXT4: 恢复异常文件系统 for crash-consistency problem



方法 2 ： Data Journaling: 提高速度

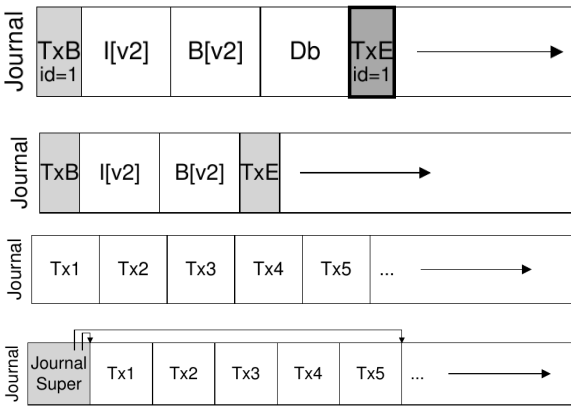
- 批处理日志更新
- 使日志有限：循环日志
- 日志超级块 journal superblock

新的更新过程

- Journal write
- Journal commit
- Checkpoint
- Free: 一段时间后，通过更新日记帐超级块将交易记录标记为空闲

细节 – 支持恢复异常 – Metadata Journaling

EXT4: 恢复异常文件系统 for crash-consistency problem

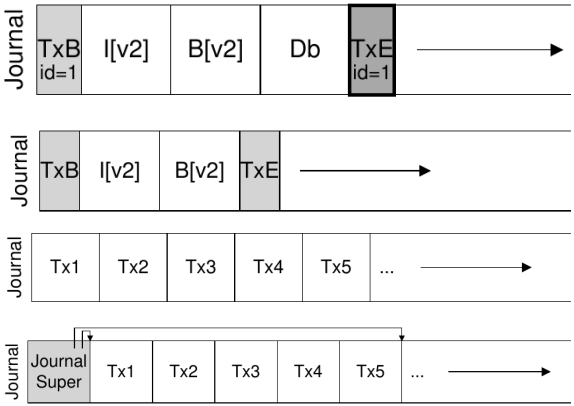


方法 2： Metadata Journaling: 进一步提高速度

- 我们什么时候应该将数据块 Db 写入磁盘?
- 事实证明，数据写入的顺序对于仅元数据的日记记录确实很重要。
- 如果在事务（包含 I [v2] 和 B [v2]）完成后将 Db 写入磁盘，这样有问题吗?

细节 – 支持恢复异常 – Metadata Journaling

EXT4: 恢复异常文件系统 for crash-consistency problem



方法 2： Metadata Journaling: 进一步提高速度

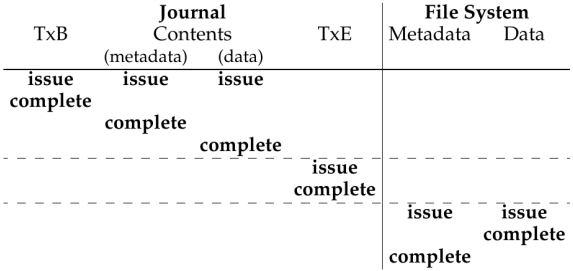
新的更新过程

- Data write
- Journal metadata write
- Journal commit
- Checkpoint metadata
- Free

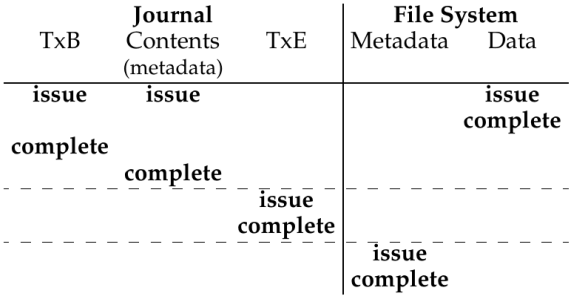
通过强制首先写入数据，文件系统可以保证指针永远不会指向垃圾数据。

细节 – 支持恢复异常 – Metadata Journaling

Data Journaling 时间线



Metadata Journaling 时间线



第十八讲：文件系统实例

第 3 节：Zettabyte File System (ZFS)

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

提纲

Ref:

- Richard McDougall, Jim Mauro, Solaris Internals:Solaris 10 and OpenSolaris Kernel Architecture, 2nd Edition, Prentice Hall, July 10, 2006, ISBN 0-13-148209-2
- ZFS: The Last Word in File Systems

What is ZFS?

ZFS is a new kind of filesystem that provides simple administration, transactional semantics, end-to-end data integrity, and immense scalability .

- Pooled storage
  - Completely eliminates the antique notion of volumes
  - Does for storage what VM did for memory
- Transactional object system
  - Always consistent on disk –no fsck, ever
  - Universal –file, block, iSCSI, swap ...
- Provable end-to-end data integrity
  - Detects and corrects silent data corruption
  - Historically considered “too expensive” –no longer true
- Simple administration
  - Concisely express your intent

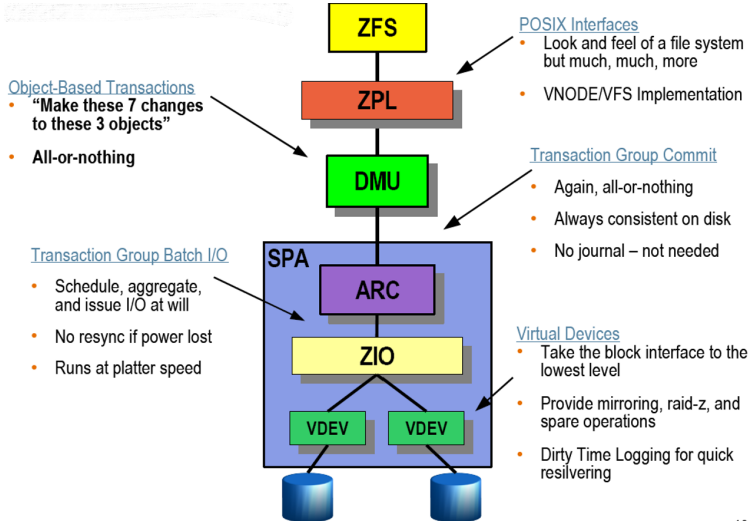
ZFS Features

- Immense capacity
  - 128bit
- Provable data integrity
  - Detects and corrects silent data corruption
- Simple administration
  - a pleasure to use

Pooled storage

- No volume
- Pooled storage
- Many file systems share pool
- And share all I/O channel in the pool

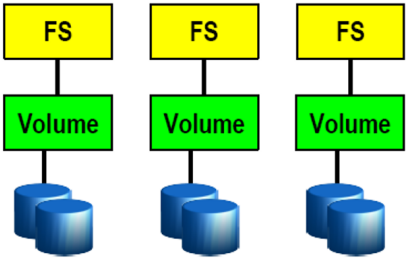
ZFS I/O Stack


ZFS
ZPL
DMU
SPA
ARC
ZIO
VDEV
VDEV
POSIX Interfaces
Look and feel of a file system but much, much, more
VNODE/VFS Implementation
Object-Based Transactions
"Make these 7 changes to these 3 objects"
All-or-nothing
Transaction Group Commit
Again, all-or-nothing
Always consistent on disk
No journal – not needed
Transaction Group Batch I/O
Schedule, aggregate, and issue I/O at will
No resync if power lost
Runs at platter speed
Virtual Devices
Take the block interface to the lowest level
Provide mirroring, raid-z, and spare operations
Dirty Time Logging for quick resilvering

FS/Volume Model vs. ZFS

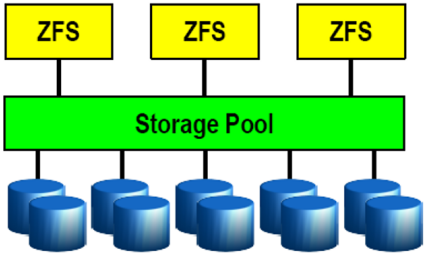
Traditional Volumes

- Abstraction: virtual disk
- Partition/volume for each FS
- Grow/shrink by hand
- Each FS has limited bandwidth
- Storage is fragmented, stranded

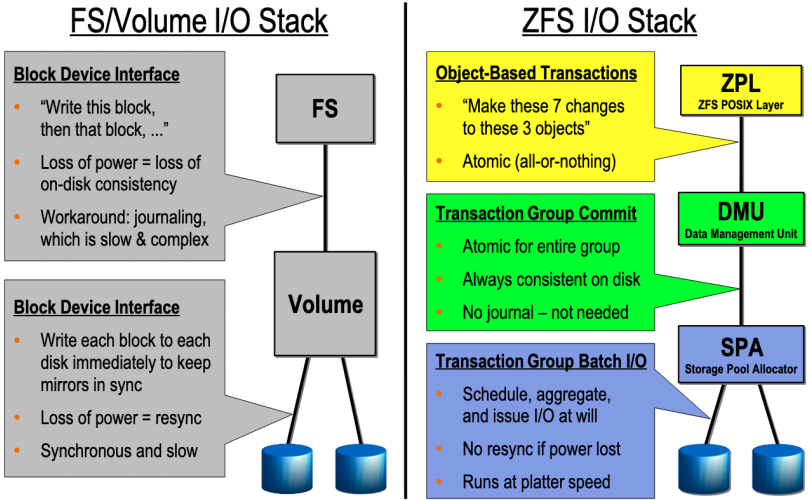


ZFS Pooled Storage

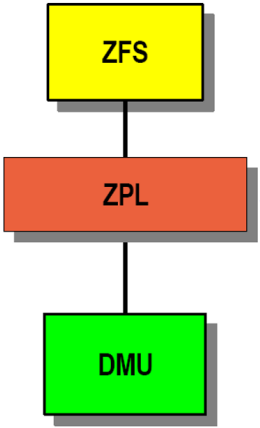
- Abstraction: malloc/free
- No partitions to manage
- Grow/shrink automatically
- All bandwidth always available
- All storage in the pool is shared



FS/Volume Interfaces vs. ZFS

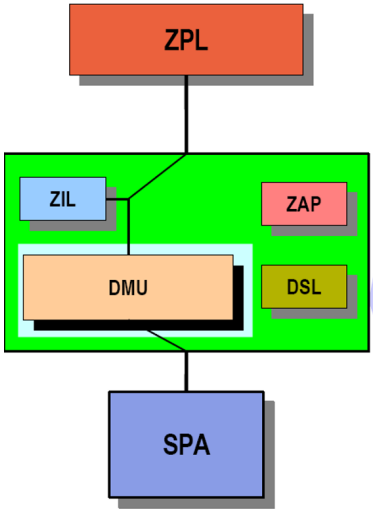


ZFL (ZFS POSIX Layer)



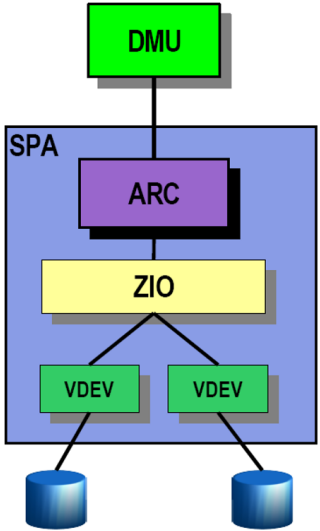
- The ZPL is the primary interface for interacting with ZFS as a filesystem.
- It is a layer that sits atop the DMU and presents a filesystem abstraction of files and directories.
- It is responsible for bridging the gap between the VFS interfaces and the underlying DMU interfaces.

DMU (Data Management Unit)



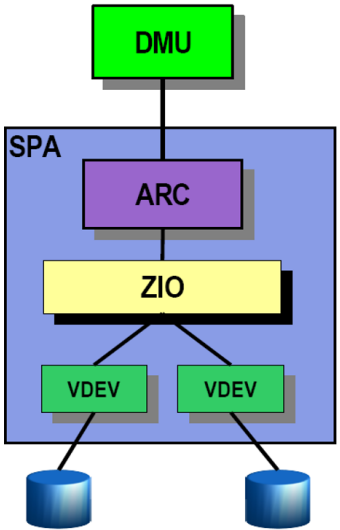
- Responsible for presenting a transactional object model, built atop the flat address space presented by the SPA.
- Consumers interact with the DMU via objsets, objects, and transactions.
- An objset is a collection of objects, where each object are pieces of storage from the SPA (i.e. a collection of blocks).
- Each transaction is a series of operations that must be committed to disk as a group; it is central to the on-disk consistency for ZFS.

ARC (Adaptive Replacement Cache)



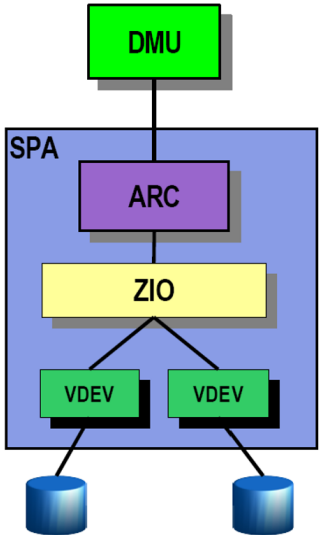
- DVA (Data Virtual Address) based cache used by DMU
- Self-tuning cache will adjust based on I/O workload
  - > Replaces the page cache
- Central point for memory management for the SPA
  - > Ability to evict buffers as a result of memory pressure

ZIO (ZFS I/O Pipeline)



- Centralized I/O framework
  - > I/Os follow a structured pipeline
- Translates DVAs to logical locations on vdevs
- Drives dynamic striping and I/O retries across all active vdevs
- Drives compression, checksumming, and data redundancy

VDEV (Virtual Devices)

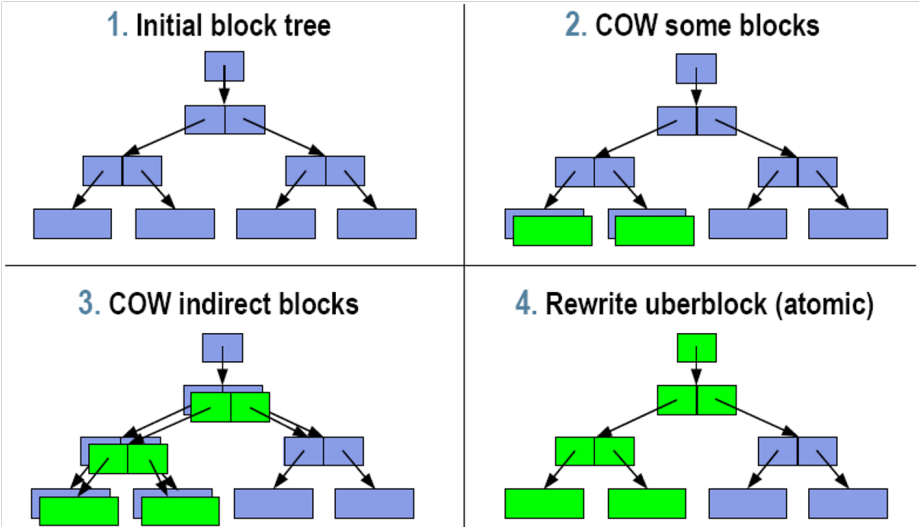


- Abstraction of devices
  - > Physical devices (leaf vdevs)
  - > Logical devices (internal vdevs)
- Implementation of data replication algorithms
  - > Mirroring, RAID-Z, and RAID-Z2
- Interfaces with the block level devices
- Provides I/O scheduling and caching

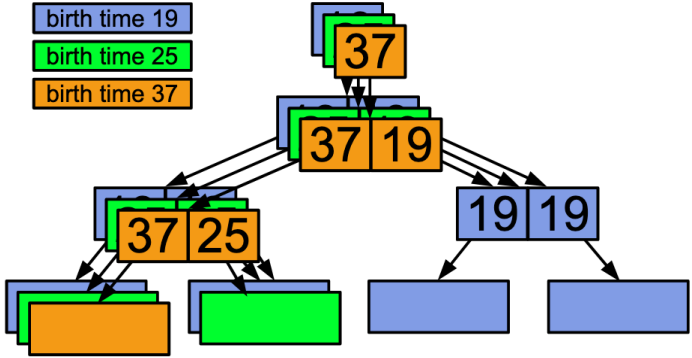
ZFS Data Integrity Model

- Everything is copy-on-write
  - Never overwrite live data
  - On-disk state always valid –no “windows of vulnerability”
  - No need for fsck(1M)
- Everything is transactional
  - Related changes succeed or fail as a whole
  - No need for journaling
- Everything is checksummed
  - No silent data corruption

Copy-On-Write Transactions



Constant-Time Snapshots

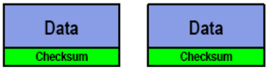


- At end of TX group, don't free COWed blocks
  - Actually cheaper to take a snapshot than not!
- The tricky part: how do you know when a block is free?

End-to-End Checksums

Disk Block Checksums

- Checksum stored with data block
- Any self-consistent block will pass
- Can't even detect stray writes
- Inherent FS/volume interface limitation

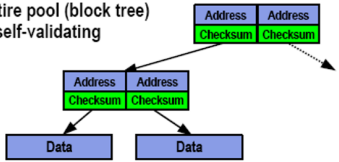


Disk checksum only validates media

- ✔ Bit rot
- ✗ Phantom writes
- ✗ Misdirected reads and writes
- ✗ DMA parity errors
- ✗ Driver bugs
- ✗ Accidental overwrite

ZFS Checksum Trees

- Checksum stored in parent block pointer
- Fault isolation between data and checksum
- Entire pool (block tree) is self-validating

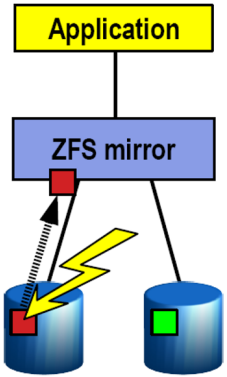


ZFS validates the entire I/O path

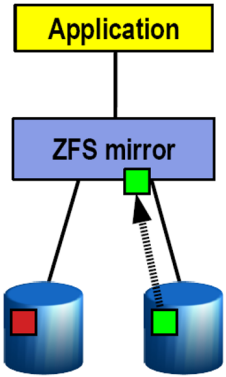
- ✔ Bit rot
- ✔ Phantom writes
- ✔ Misdirected reads and writes
- ✔ DMA parity errors
- ✔ Driver bugs
- ✔ Accidental overwrite

Self-Healing Data in ZFS

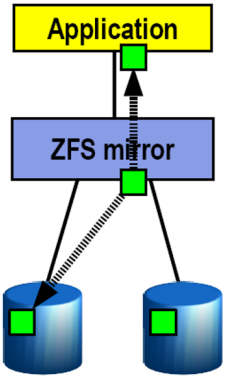
1. Application issues a read. ZFS mirror tries the first disk. Checksum reveals that the block is corrupt on disk.



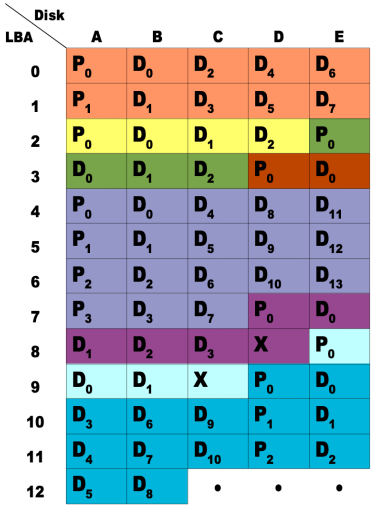
2. ZFS tries the second disk. Checksum indicates that the block is good.



3. ZFS returns good data to the application and repairs the damaged block.



RAID-Z

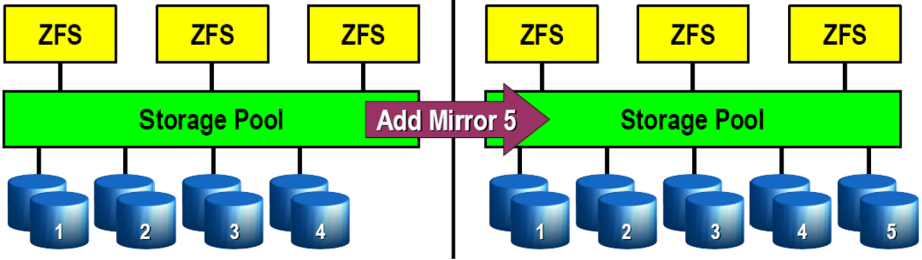


- Dynamic stripe width
  - Variable block size: 512–128K
  - Each logical block is its own stripe
- All writes are full-stripe writes
  - Eliminates read-modify-write (it's fast)
  - Eliminates the RAID-5 write hole(no need for NVRAM)
- Both single and double parity
- Detects and corrects silent data corruption
  - Checksum-driven combinatorial reconstruction
- No special hardware–ZFS loves cheap disks

Dynamic Striping

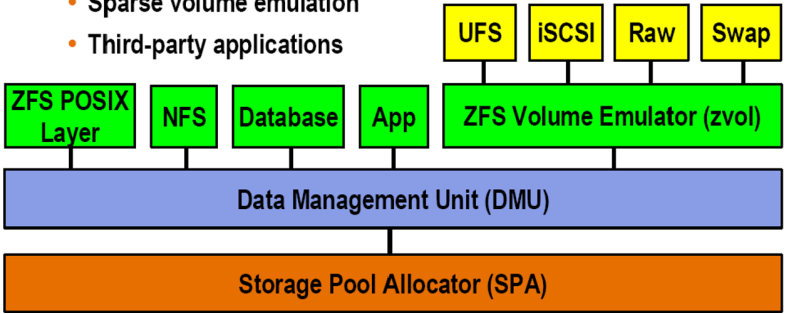
- Writes: striped across all four mirrors
- Reads: wherever the data was written
- Block allocation policy considers:
  - Capacity
  - Performance (latency, BW)
  - Health (degraded mirrors)

- Writes: striped across all five mirrors
- Reads: wherever the data was written
- No need to migrate existing data
  - Old data striped across 1-4
  - New data striped across 1-5
  - COW gently reallocates old data



Object-Based Storage

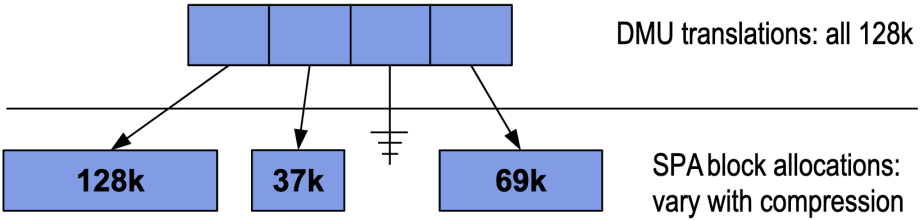
- **DMU is a general-purpose transactional object store**
  - Filesystems
  - Databases
  - Swap space
  - Sparse volume emulation
  - Third-party applications



Variable Block Size

- No single block size is optimal for everything
  - Large blocks: less metadata, higher bandwidth
  - Small blocks: more space-efficient for small objects
  - Record-structured files (e.g. databases) have natural granularity; filesystem must match it to avoid read/modify/write
- Per-object granularity
  - A 37k file consumes 37k –no wasted space
- Enables transparent block-based compression

Built-in Compression



- Block-level compression in SPA
- Transparent to all other layers
- Each block compressed independently
- All-zero blocks converted into file holes
- LZJB and GZIP available today; more on the way