DE LA RECHERCHE À L'INDUSTRIE

cea

www.cea.fr

Local Filesystems

Jacques-Charles Lafoucriere

ENSIIE | ASE

Local File Systems Lecture Outline

Day 1

- Definitions
- Storage Media
- Linux VFS
- FUSE
- DOS

Day 2

- FFS
- LFS
- ExtN
- NOVA
- LTFS

File system: Definitions

What is a File System?

- Storage medium is a long stream of bytes
- Data need to be organized
  - Split in parts
  - Named
- FS is
  - An organization of data to achieve some goals
    - Speed, flexibility, integrity, security
  - A software used to control how data is stored and retrieved
  - Can be optimized for some media
    - ISO 9660 for optical discs
    - tmpfs for memory

FS concepts

Space management

- Allocate/Free space in blocks
- Organize Files and Directories
- Keep track of which media belong to which file and which space is free

- FS fragmentation
  - Space allocated to file is not contiguous
  - Imply lots of jump
  - Decrease performance
  - Come from file write/remove when media is close to full

FS layers is OS

Logical FS

- API for file operations

Virtual FS

- Common interface to allow support of multiple physical FS
- Implement shared services for all Physical FS

Physical FS

- Manage the physical operations on the device
- Implement all the logic specific to the FS

FS concepts

Filenames

- Identify a file
- May have some limit
  - Length, Case sensitive or not
  - Forbidden char (like directories separator)
  - Today most FS support Unicode

Directories (or folders)

- Used to group files
- Can be flat or hierarchical
- May have some limit
  - Same as filenames
  - Total length of Path

Symlinks/Junctions

- Specific file
- Used to reference “transparently” an object

FS concepts

Metadata

- File/Directories attributes
  - Size
  - Creation/Modification/Access/MD change times
  - Owners, groups
  - Security
  - Format
- Global MD
  - free space bitmap
  - block availability map
  - bad sectors
- Update policies and coherency will have a huge impact on performance

FS concepts

Security

- Goal is to prevent access/modification from some users/groups
- Permissions bit field or ACL

Integrity

- Goal is to warranty data read is data written
- Protect against
  - HW failure
  - Unexpected stops
  - SW bugs
  - Silent corruption (bit flip)

FS concepts

User Data

- Main purpose of FS is to manage user data
  - Store, retrieve, update
- Access semantic can change from one FS to another one
  - Key-Value
  - Posix (byte stream)
  - OpenVMS (record oriented)
  - Update granularity

FS Concepts

Snapshot

- A frozen image of as FS at a given time
  - Totally coherent
    - Quick instant backups can be used for a long copy
    - Live dumps
- How: Do copy on write for all changes
  1. Make change in new place
  2. Duplicate old ref
  3. Change ref to new place
  - Use of old ref give the frozen view
- Consequences
  - Consume space
  - Increase complexity of updates
- A RW snapshot is named a clone

FS Concepts

Quotas

- To control resource allocation
- Limit number of files and number of disk blocks per user and/or per group and/or per project
- 2 limits
  - Soft: can be cross during a grace period
  - Hard: cannot be crossed
- Limits are checked during allocation process

File locking

- To help controlling file sharing
- Different type: RO, RW, EX
- Explicit on many FS (Unix like)
- Implicit on some (NTFS)

FS Concepts

Deduplication

- To optimize resource allocation
- Compression like feature
- Identify equal datasets
- Use a single copy
  - All ref are done to the same block (hidden sharing)
  - Ref are changed at the first write

FS concepts

Using a FS

- Administration tools
  - FS creation
  - FS check
  - Quotas Check
- STD Libraries
  - Implement access semantic
- FS API
  - Specific to a FS

Type of local FS

Disk FS (HDD)

- FAT, ExtN, FFS, LFS, Versioning FS

Optical disk FS (CDROM, DVD, Blu-Ray)

- ISO9660, Universal Disk Format

Flash FS (SSD, Flash card)

- NOVA

Memory FS

- tmpfs

Tapes FS

- LTFS (Linear Tape FS)

Storage Media

Types of medium

Hard Disk Drive

- Rotating media
- Magnetic recording

Flash memory

- No movement
- Electronic recording

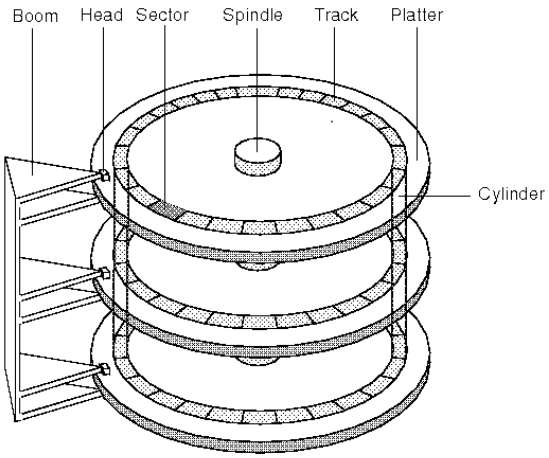
Tapes

- Linear recording
- Magnetic recording

Hard Disk Drive

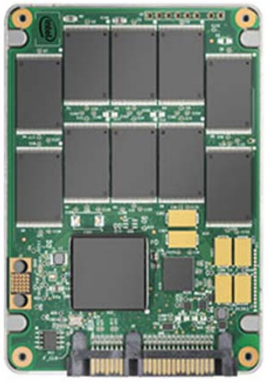
- Rotating media
- Magnetic recording
- Date recorded on concentric circles: tracks
- Allocation vs geometry
  - Has an impact on performances

- HDD mechanical components



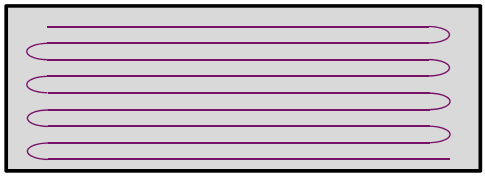
Solid State Drive

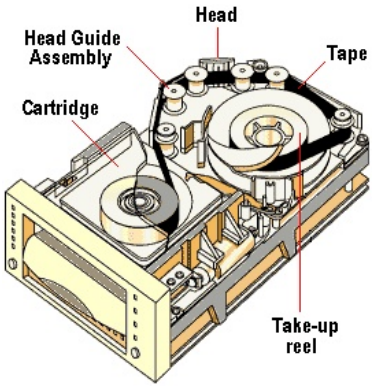
- Only chips, no movement
- Electronic recording based on transistors
- Uniform access
- Flash media needs a special management
  - Write once (no update, erase before write)
  - Chips has a pre-defined write count
  - Erase by block only
  - Garbage collecting
- Required help from FS to keep performance



Tapes

- Linear tape media
- Tape length > 800 m
- Magnetic recording
- Serpentine recording





Devices Comparison

	HDD	SSD	TAPE
Size	++	+	+++
Access Time	++ (10 ms)	+++ (few ms)	- (minutes)
Bandwidth	++ few 100 MB/s	+++ many 100 MB/s	+ few 100 MB/s when streaming
Random Use	++	+++	-
Sequential Use	+++	+++	++ (wo stop)
Power Usage	++ (10W)	+ (> 10W)	+++
Life Time	++	+ (linked to usage)	+++

No Universal FS ☹

LINUX VFS

Linux VFS

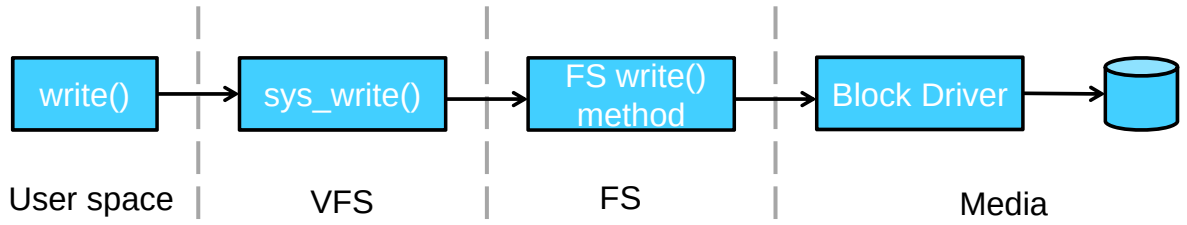
Virtual Filesystem

- Kernel subsystem
- Implements the FS related interfaces provided to user space
- Allows
  - FS coexistence
  - STD interfaces for applications
- Seems “normal” but was not the case previously
  - Eg: in MSDOS all access to a non native FS required special tools
  - Only modern OS provide VFS layers
  - With VFS the same tool can copy files between NFS, ext3: tool ignore FS type
- Hides FS internals to applications

Linux VFS: API stack

Write example

- write(file_descriptor, &buffer, length)



Linux VFS

API

- Interfaces
- Data structures
- Common concepts: files, directories, …

Follow Unix FS abstractions

- File, directory, mount point, inode
- FS are mounted at a specific mount point in a global hierarchy: a namespace
  - All mounted FS appear in a single tree

Linux VFS Objects

VFS is object oriented

- Object = data + methods
- Methods apply on parent object
- Many methods can use generic functions for basic features

Superblock object

- Represent a specific mounted FS

Inode object

- Represent a specific file

Dentry object

- Represent a specific directory (eg a single component of a path)

File object

- Represent an open file associated with a process

Superblock Object

- Implemented for each mounted FS
- Store informations describing the mounted FS
- Usually match the local FS superblock
- Declared in <linux/fs.h>
- Code to create/manage/destroy SB in fs/super.c

- Contains
  - Block size
  - Max file size
  - Type, magic number
  - Flags
  - Lists: dirty inodes, writebacks, anonymous dentries, files, …
  - Pointer to methods

```
struct super_block {
          struct list_head    s_list;
          dev_t               s_dev,
          ..........
};
```


Superblock Methods

```
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);
        void (*dirty_inode) (struct inode *, int flags);
        int (*write_inode) (struct inode *, struct writeback_control *wbc);
        int (*drop_inode) (struct inode *);
        void (*evict_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        int (*sync_fs)(struct super_block *sb, int wait);
        int (*freeze_super) (struct super_block *);
        int (*freeze_fs) (struct super_block *);
        int (*thaw_super) (struct super_block *);
        int (*unfreeze_fs) (struct super_block *);
        int (*statfs) (struct dentry *, struct kstatfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*umount_begin) (struct super_block *);
        int (*show_options)(struct seq_file *, struct dentry *);
        int (*show_devname)(struct seq_file *, struct dentry *);
        int (*show_path)(struct seq_file *, struct dentry *);
        int (*show_stats)(struct seq_file *, struct dentry *);
        ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
        ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
        struct dquot **(*get_dquots)(struct inode *);
        int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
        long (*nr_cached_objects)(struct super_block *, struct shrink_control *);
        long (*free_cached_objects)(struct super_block *,struct shrink_control *);
};
```

Inode Object

- Implemented for each file/directory manipulated by the kernel
- Store information needed to manage a file or directory
- Usually match the local disk inode
- Declared in <linux/fs>
- Associated code is in FS

- Contains
  - Ref counters
  - Unix metadata
  - Pointer to methods

```
struct inode {
          …
          struct list_head    i_list;
          struct list_head    i_dentry;
          unsigned long       i_ino;
          …
};
```


Inode Methods

```
struct inode_operations {
        struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
        const char * (*get_link) (struct dentry *, struct inode *, struct delayed_call *);
        int (*permission) (struct inode *, int);
        struct posix_acl * (*get_acl)(struct inode *, int);
        int (*readlink) (struct dentry *, char __user *,int);
        int (*create) (struct inode *,struct dentry *, umode_t, bool);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,umode_t);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
        int (*rename) (struct inode *, struct dentry *, struct inode *, struct dentry *, unsigned int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start, u64 len);
        int (*update_time)(struct inode *, struct timespec *, int);
        int (*atomic_open)(struct inode *, struct dentry *, struct file *, unsigned open_flag,
                           umode_t create_mode, int *opened);
        int (*tmpfile) (struct inode *, struct dentry *, umode_t);
        int (*set_acl)(struct inode *, struct posix_acl *, int);
};
```

Dentry Object

- Implemented for each file/directory manipulated by the kernel
- Store information needed to manage a path
- No match to the local disk structures
- Declared in <linux/dcache.h>
- Associated code is in fs/dcache.c
- Contains
  - Ref counters
  - Ref to position in path
  - Pointer to methods

```
struct dentry {
        atomic_t            d_count;
        struct inode        d_inode;
        struct list_head    d_chid;
        struct list_head    d_subdirs;
        …
};
```

- 3 states: used (d_count > 0), unused (d_count = 0), negative (d_inode = NULL)

Dentry Methods

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, unsigned int);
        int (*d_weak_revalidate)(struct dentry *, unsigned int);
        int (*d_hash)(const struct dentry *, struct qstr *);
        int (*d_compare)(const struct dentry *, unsigned int, const char *, const struct qstr *);
        int (*d_delete)(const struct dentry *);
        int (*d_init)(struct dentry *);
        void (*d_release)(struct dentry *);
        void (*d_prune)(struct dentry *);
        void (*d_iput)(struct dentry *, struct inode *);
        char *(*d_dname)(struct dentry *, char *, int);
        struct vfsmount *(*d_automount)(struct path *);
        int (*d_manage)(struct dentry *, bool);
        struct dentry *(*d_real)(struct dentry *, const struct inode *, unsigned int);
};
```

File Object

- Implemented for each file open by the kernel
- Multiple objects can exist for a single file
- Store information needed to represent a file from a process view
- No match to the local disk structures
- Declared in <linux/fs.h>
- Associated code is in fs/file.c

- Contains
  - Ref counters
  - Offset
  - Pointer to methods

```
struct file {
        struct list_head    f_list;
        struct dentry       f_dentry;
        atomic_t            f_count;
        …
};
```


File Methods

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
        int (*iterate_shared) (struct file *, struct dir_context *);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
        unsigned (*mmap_capabilities)(struct file *);
        ssize_t (*copy_file_range)(struct file *, loff_t, struct file *, loff_t, size_t, unsigned int);
        int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t, u64);
        ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *, u64);
};
```

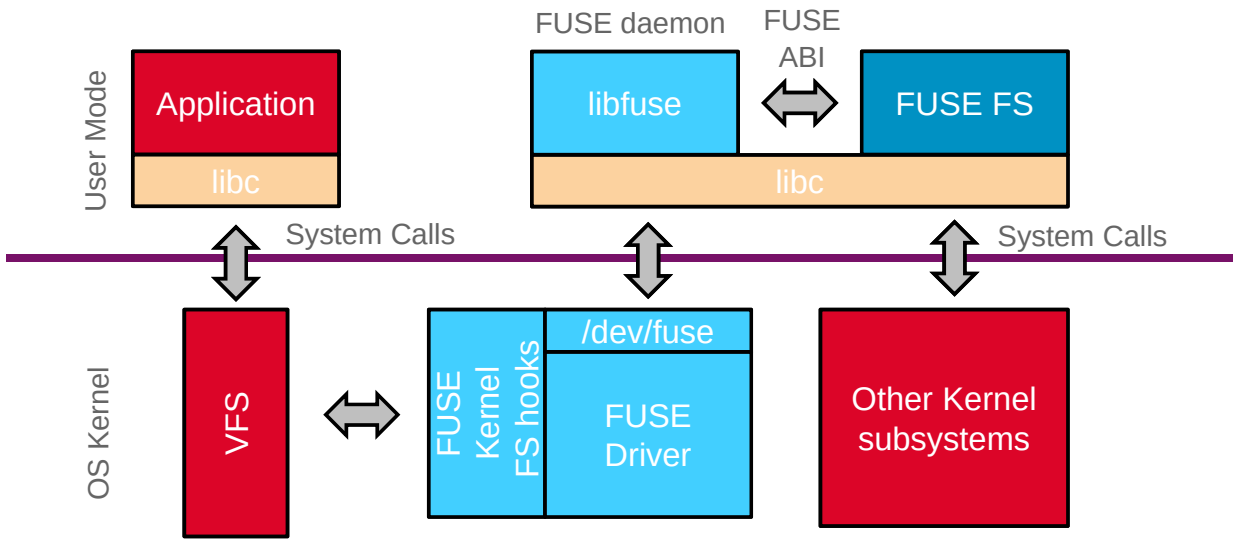
FUSE

FUSE: Why?

2 places to run a FS

- Kernel Space
  - Best performance for local access
- User Space
  - Stable/Documented interfaces
  - Portable FS become possible: Linux/BSD/MacOS
  - New languages usable: python
  - Debugging easier
  - Networking, cache management, … are easier
  - Can run as unprivileged user (good for security)
  - Better resilience: FS crash does not crash system

FUSE Architecture



Developing a FUSE FS

Rich interface

- 40 possible operations
- Many operations have good default
  - Minimum of 4
- Directory Operations
- File Operations
- Metadata Operations
- Locking, Init

```
static struct fuse_operations simple_oper = {
        .getattr        = simple_getattr,
#if FUSE_VERSION > 25
        .readdir        = simple_readdir,
#else
        .getdir         = simple_getdir,
#endif
        .open           = simple_open,
        .read           = simple_read,
};
```

Defining Operations

- C: function pointers in fuse structure
- Python: method on subclass of fuse.Fuse

FUSE Operations

Directory Operations

File Operations

Metadata Operations

Locking, Init

FUSE Operations

```
int (*getattr) (const char *, struct stat *, struct
fuse_file_info *fi);
int (*readlink) (const char *, char *, size_t);
int (*mknod) (const char *, mode_t, dev_t);
int (*mkdir) (const char *, mode_t);
int (*unlink) (const char *);
int (*rmdir) (const char *);
int (*symlink) (const char *, const char *);
int (*rename) (const char *, const char *, unsigned int flags);
int (*link) (const char *, const char *);
int (*chmod) (const char *, mode_t, struct fuse_file_info *fi);
int (*chown) (const char *, uid_t, gid_t, struct fuse_file_info
*fi);
int (*truncate) (const char *, off_t, struct fuse_file_info
*fi);
int (*open) (const char *, struct fuse_file_info *);
int (*read) (const char *, char *, size_t, off_t, struct
fuse_file_info *);
int (*write) (const char *, const char *, size_t, off_t, struct
fuse_file_info *);
int (*statfs) (const char *, struct statvfs *);
int (*flush) (const char *, struct fuse_file_info *);
int (*release) (const char *, struct fuse_file_info *);
int (*fsync) (const char *, int, struct fuse_file_info *);
int (*setxattr) (const char *, const char *, const char *,
size_t, int);
int (*getxattr) (const char *, const char *, char *, size_t);
```

```
int (*listxattr) (const char *, char *, size_t);
int (*removexattr) (const char *, const char *);
int (*opendir) (const char *, struct fuse_file_info *);
int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t,
struct fuse_file_info *, enum fuse_readdir_flags);
int (*releasedir) (const char *, struct fuse_file_info *);
int (*fsyncdir) (const char *, int, struct fuse_file_info *);
void *(*init) (struct fuse_conn_info *conn, struct fuse_config
*cfg);
void (*destroy) (void *private_data);
int (*access) (const char *, int);
int (*create) (const char *, mode_t, struct fuse_file_info *);
int (*lock) (const char *, struct fuse_file_info *, int cmd,
struct flock *);
int (*utimens) (const char *, const struct timespec tv[2],
struct fuse_file_info *fi);
int (*bmap) (const char *, size_t blocksize, uint64_t *idx);
int (*ioctl) (const char *, int cmd, void *arg, struct
fuse_file_info *, unsigned int flags, void *data);
int (*poll) (const char *, struct fuse_file_info *, struct
fuse_pollhandle *ph, unsigned *reventsp);
int (*write_buf) (const char *, struct fuse_bufvec *buf, off_t
off, struct fuse_file_info *);
int (*read_buf) (const char *, struct fuse_bufvec **bufp, size_t
size, off_t off, struct fuse_file_info *);
int (*flock) (const char *, struct fuse_file_info *, int op);
int (*fallocate) (const char *, int, off_t, off_t, struct
fuse_file_info *);
```

Using FUSE

To mount

- ./my_new_fs DIR

To unmount

- fusermount -u DIR

DOS

DOS FS

Origin

- Designed for MSDOS
- Simple FS
  - Initially designed for diskettes (360KB)
  - Available on any platform
  - Used in many small devices: camera (DCF)

Characteristics

- Simple filename format: 8.3 (also support of long file names)
- Initially flat namespace
  - Directories support with DOS2, path up to 64 characters
- Based on File Allocation Table
  - FAT(12), FAT16, FAT32

DOS layout

Reserved sectors

- Bootstrap sector: disk characteristics, # of copies of FAT

File Allocation Tables

- # copies, DOS uses 1st, if bad try to use another; All copies are always updated
- One entry for each logical block in the volume
  - Free or the logical block number of the next logical block in the file

Root directory

- Contain numbers of first cluster of each file in that directory
- Each file is a cluster chain

Data Area

Hidden sectors

Directory Entry Format

```
byte(s) contents
------- ------------------------------------------------
0-7     file name or first 8 characters of volume name
        (0xE5/0x05 = deleted, 0x00 last entry)
8-10    file extension or last 3 characters of volume name
11      attribute byte (type of entry)
12-21   unused
22-23   time
24-25   date
26-27   number of first cluster
28-31   number of bytes in file, or zero for subdirectory or volume label
```


FFS

FFS

Origin

- Unix File System (UFS) or Berkeley Fast File System (1984)
  - Used by many Unix like OS
- Used in many academic research
- Sources available and highly documented
- Has all the basic concepts found in all FS
- Designed to optimize HDD access
  - try to localize associated data blocks and metadata in the same cylinder group and, ideally, all of the contents of a directory in the same or nearby cylinder group

Availability

- Berkeley Software Distribution OS's: FreeBSD, OpenBSD, NetBSD, …
- Linux (RO)
- Old MacOS

FFS Design

Boot blocks

- Reserved for OS, not used by FS

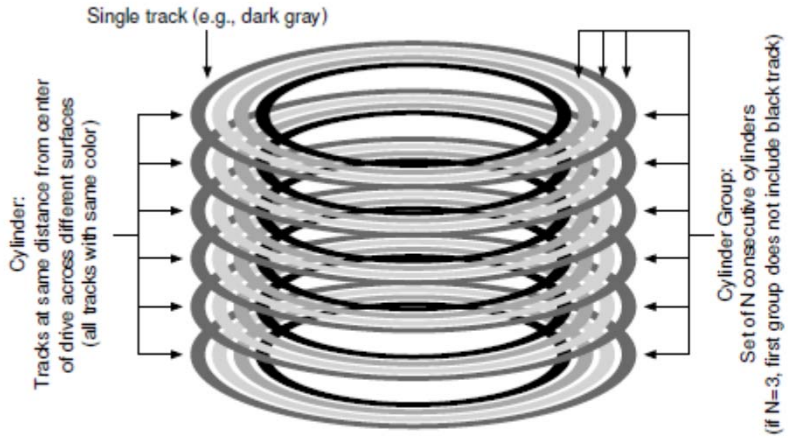
Superblock

- Magic number, FS geometry, statistics and tuning parameters

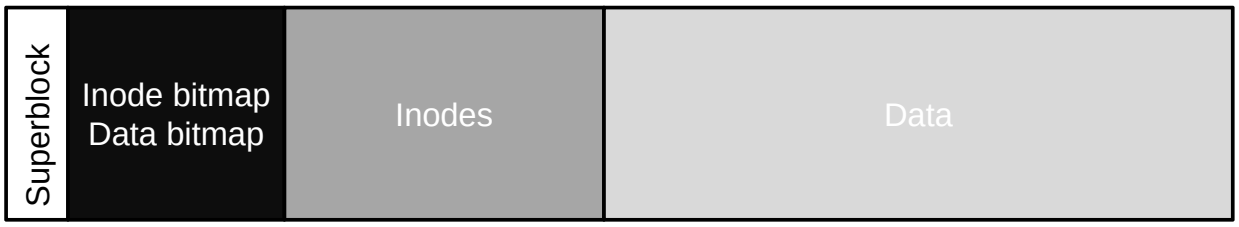
Collection of cylinder groups

- Copy of SB
- CG header: statistics, free lists
- Inodes
- Data blocs
- CG brings locality, avoid disk seek and reduce fragmentation
  - /!\ no more doable/needed for modern disks
  - Recent FS stile use the locality concept but wo geometry knowledge

Cylinder Group Concept



Cylinder Group



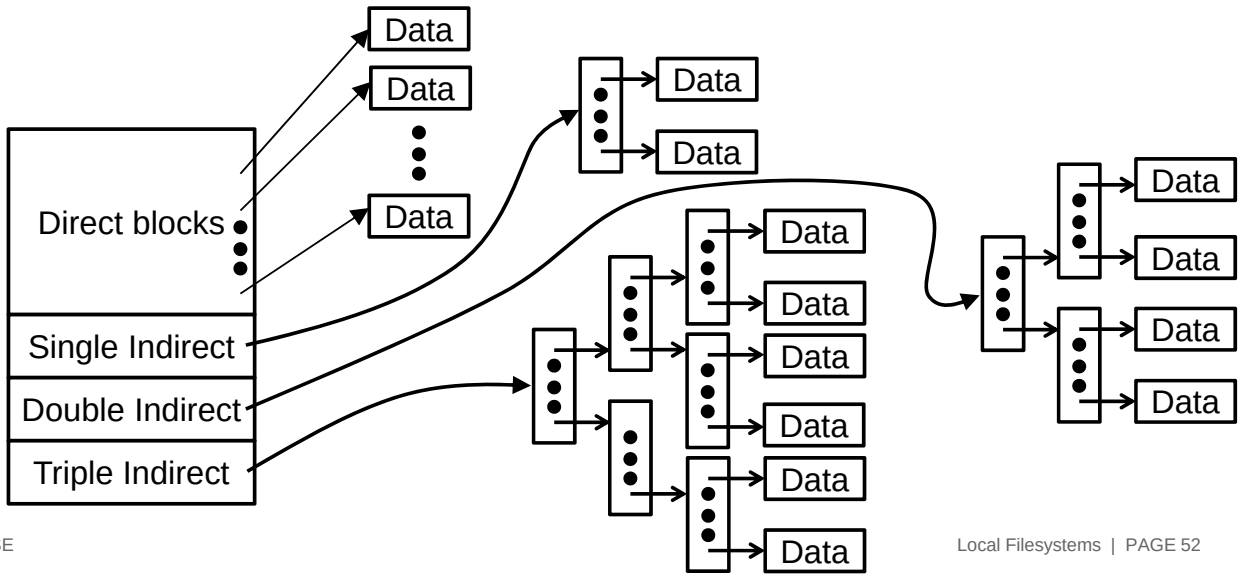
- Bitmaps are an efficient way to manage free space in a FS
  - Easy to find a large chunk of free space (pattern matching)

File Creation

- Allocate an inode
  - Inode Bitmap
- Allocate data
  - Data bitmap
- Add file to directory
  - Update directory content
    - Read/Modify/Write, may need a new block
  - Update inode directory (size, count, times, …)
- Update global counters

- Allocation policy has a huge impact on performances
  - Creation and Access

Inode Data Blocks Structure



Inode

- Type
- Access mode
- Inode number
  - root dir = 2, lost+found = 3
- Times
- Size
- Number of blocks
- Number of directories entry pointing to the file
- Generation number (random number used for inode reallocation)
- Block size
- Size of extended attributes
- Data blocks pointers

- No filename, because filename is in directory
  - Allow multiple names (hardlink)

Directory

- A chunk: size chosen to allow a single write to disk -> atomic updates
- Inode number
- Size of entry
  - Also used to find next entry
- Type of entry
- Length of name

- To create a new entry
  - FS check if name exist
  - During the check, it look for a free space large enough
  - Do compaction if needed
  - If name does not exist, creates it

Soft Updates

Origin

- FS must maintain a global integrity of it's metadata even in case of crashes or power lost
- Many FS use synchronous writes
  - Impact on performances
- Idea
  - Group each of the dependent updates as an atomic operation with write ahead logging
    1. Describe
    2. Change
    3. Commit
  - Identified operations
    - file/dir creation, file/dir removal, file/dir rename, block allocation, indirect block manipulation, free map management

FFS Summary

- Introduced many new ideas
  - Storage-aware layout
  - Soft updates
- Added a number of new features
  - Long file names
  - Symbolic links
  - Atomic rename (new syscall)
- Still today many file systems like extX use ideas from FFS

LFS

Log-structured FS

Origin

- In 90's, more memory was available for caching
  - So main disk traffic has to be optimized for writes
- Random access are bad
  - Use sequential access
- Classical FS (FFS) perform poorly on many small file creation load
- FS are not RAID aware
  - Use of small blocs

Availability

- BSD systems

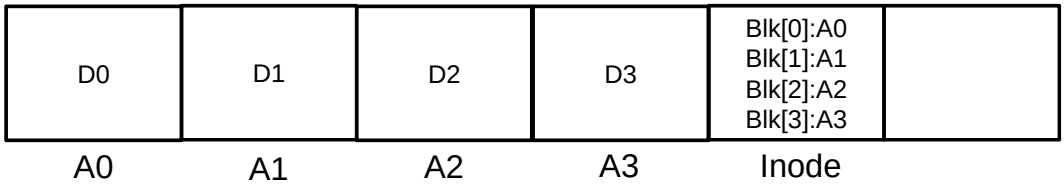
LFS design

Idea

- All updates are buffered in memory segments
- When segment is full, write to disk to an unused place
  - Never overwrite
  - Always large sequential writes

Problem: How to transform all updates into sequential writes?

- Buffer all updates



How to find Inodes?

No more standard location

- Inodes are spread over all the disk surface
- Add an inodemap
  - Indirection map
  - Keep a reference to inode location
  - Put after the last update to avoid hotspot
  - Avoid directory update when inode is updated

How to find Imap?

- Use a fixed place, the checkpoint region
- Update it rarely

Space Management

Update only is nice but how to reuse old blocks?

- Garbage Collection
  - Find unused blocks
  - Policy to decide when to scan/purge segments
- Block Liveness
  - Store inode # and offset in block header
  - Use it to compare with inode description
- Policy
  - Based on segment temperature
- Garbage collection is difficult and is the main downside of this approach

EXT2/3/4

ExtN

Origin

- Ext was the first Linux FS after use of Minix (in 1992)
  - Solve 2 Minix limitation: 64MB limit and 14 chars filenames
- Ext2 was designed following the principles of FFS (1993)
- Ext3 add (2001)
  - Journaling
  - Online FS growth
  - Htree indexing for larger directories

ExtN

- Ext4 add (2008)
  - Large file system
  - Extents support
  - Persistent pre-allocation, delayed allocation
  - Unlimited number of subdirectories
  - Journal checksumming
  - Metadata checksumming
  - Multiblock allocators
  - Improved timestamp

Ext4 Concepts

Very similar to FFS

- But much less documented …

Journaling

- 3 levels of security
  - Journal
    - All updates go to journal before being committed
  - Ordered
    - Only metadata updates go to journal
    - Data are written before metadata commit
  - Writeback
    - Only metadata updates go to journal
    - No rules for data

Ext4 Concepts

Large File System

- FS up to 1 EiB, files up to 16 TiB

Extents

- Replace fixe size blocks by offset/length
- Tree index for more than 4 extents

Unlimited number of subdirectories

- Use of HTree indexes

Improved timestamp

- Timestamp in nanoseconds

ZFS

ZFS

Origin

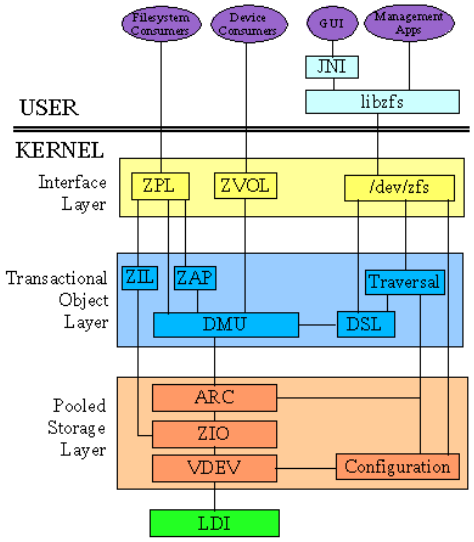
- Zettabyte Filesystem
- Recent design (2001-2004)
- Never overwrite existing data
  - Perfect for free snapshots and clones
- All changes are made in memory and flushed in a coherent state on disk (checkpoint)
  - No corruption risk, move from a coherent state to a coherent state
- Everything is checksumed
- Merge device management (volume manager) and file system layers

Availability

- Solaris
- BSD
- Linux through third party

ZFS modules

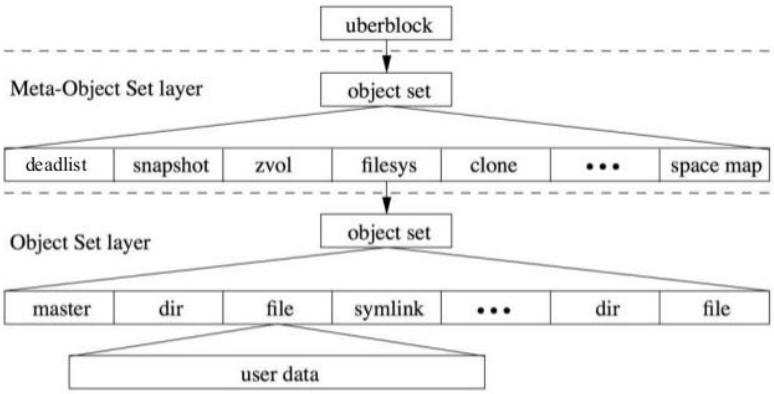
- Zfs Posix Layer
- Zfs Attribute Processor
- Data Management Unit
- Zfs Intent Log
- Dataset and Snapshot Layer
- Storage Pool Allocator
- Zfs VOLume
- Zfs I/O
- RAIDZ
- Virtual DEVice
- Adaptive Replacement Cache



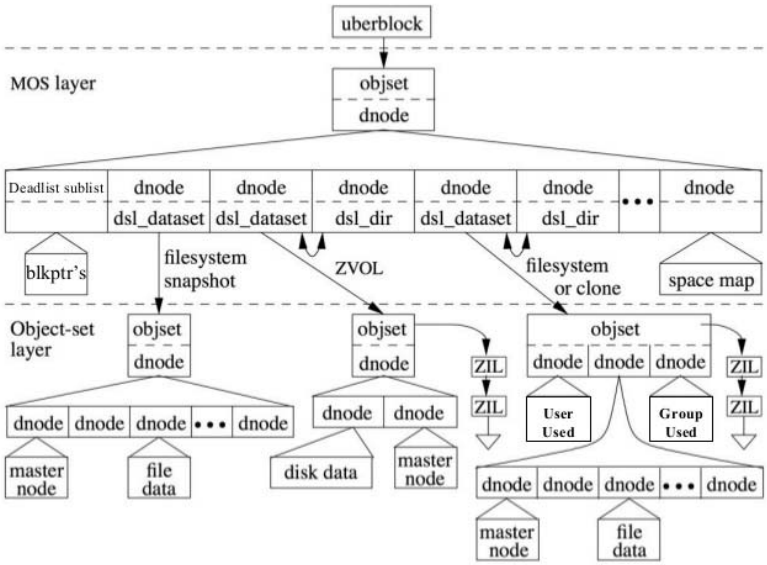
ZFS Organization

- ZFS creates pool of space where allocation is done on the flight
  - Space can be moved between FS as needed
- Uberblock is at the top
  - Points to an array of meta-objects
    - FS, snapshots, clones,, ZVOL, space map
- ZFS object metadata are stored in a dnode
  - Managed by DMU
  - Generic (file, dir, clones, zvols, …)
  - Use a fixed number of levels depending of the size
    - When dnode is too small, it is promoted to a larger one
- Posix metadata are stored in a znode
  - Managed by ZPL
  - Associated to a dnode

ZFS Organization

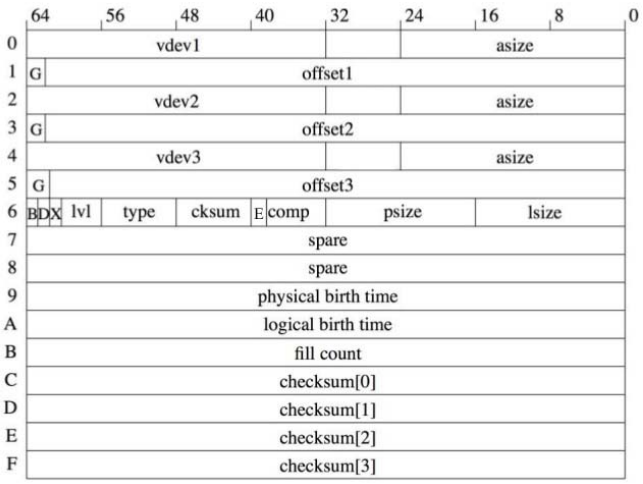


ZFS Structure



ZFS Organization

- ZAP objects stores a Key-Value hastable
  - Directories
    - Name to object number
- ZFS block pointer
  - 128 Bytes
  - Pointers to up to 3 copies
    - Device, offset, size
  - Checksum
- Every block is checksumed



ZFS Block Layers

RAIDZ

- Support variable-size stripe
- After failure, only used space is rebuild

Block Allocation

- Space allocation is handled by SPA
- Use a space map in the MOS
- Blocks are grouped in fixe-sized groups
  - Defined by base/length (like extend)

Freeing Blocks

- No need of garbage collector
- Block are explicitly released by comparing birth date to youngest snapshot

ZFS Design Tradeoffs

Disk rebuild

- Faster if disk pools are less than half full

Partial writes

- No overwrite, so avoid RAID write-hole problem

Better space management

- Shared between FS

Tight integration

- Integrity, snapshots benefit to all components
- Generic objects.

Performances

- Need free space to work well (25%)

Memory consumption

- ZFS caches in ARC require more memory
- Additional copies to kernel memory cache

ZFS

Evolution

- Licence issue seems still prevent Linux integration
- Community development very active
- Move away from Solaris

- Ideas are taken by others wo the licence issue
  - Btrfs
    - Designed from scratch
    - Linux kernel compliant
    - Less mature

NOVA

NOVA

Origin

- Persistent memory availability
  - Fast, byte-addressable, persistent
  - 2 approaches: new FS or extension of existing FS
- Research FS

Availability

- Linux

NOVA FS Concepts

Kernel integration

- Does not use block layer
- Use storage mapped into kernel's address space
  - Direct access to storage
- Big simplification
  - no request coalescing
  - no queue management
  - no prioritization of requests

NOVA FS Structure

Superblock

- Top level structure
- Statistics
- Counters
- Inode table
  - A set of per-CPU arrays
  - Any CPU can allocate new inodes without having to take cross-processor locks
- Free Space table
  - Red-Black tree on each CPU
- Both structure are managed in memory and flushed at unmount
  - Can be rebuild from device scan

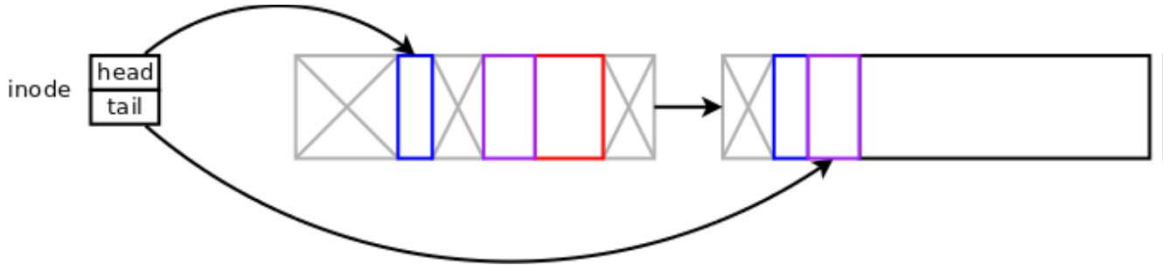
NOVA Inode design

Physically

- A pair of pointers

Logically

- A log describing the changes that have been made to the file



NOVA Inode Log Entry

Inode Records

- The attributes of the file have been changed
- An entry has been added to a directory
- A link to the file was added
- Data has been written to the file

NOVA Data Writing

Empty file

- Allocate the needed memory from the per-CPU free list
- Copy the data into that space
- Append an entry to the inode log
  - New length of the file
  - Pointer to the location in the array where the data was written.
- Atomic update of the inode tail pointer

NOVA Data Writing

Data overwriting: COW

- Allocate new memory for the new data
- Copy data from the old pages into the new
- New data is added
- New entry is added to the log pointing to the new pages
- Update the tail pointer
- Invalidate old log entry for those pages
- Free or reuse old data pages

On disk format

- As a set of instructions that, when followed in order (and skipping the invalidated ones) will yield a complete description of the file and its layout in memory

NOVA Limitations

Still under development

- x86-64 only support
- Cannot move FS from one server to another if CPU # is different
- No ACL
- No Quotas
- No fsck

LTFS

LTFS: Linear Tape File System

Origin

- Tapes are a long stream of bytes
- Access is sequential
- Access need start/stops, tape is not always in movement like HDD
- LTFS is a SNIA standard
  - Allow tape portability between OS
  - Designed for tapes
  - Is not linked to a tool like tar, dump, cpio, …

Availability

- Linux
- Windows
- MacOS

LTFS Concepts

- Tape is divided in volumes
- Volume contains
  - Data files
  - Metadata files
- Volume completely describe the directory and file structures
  - Self contained

- LTFS can be mounted
- Files can be written to / read from volume with std OS operations

LTFS Format

Volume

- 2 partitions: 1 index partition + 1 data partition

Partition

- Label construct: identify volume (volume label + FM + LTFS label + FM)
- Content area
  - Index or Data
  - Always finished by an index

Index

- FM + set of sequential logical blocks of fixed size + FM
- XML structure
- References to other index (chained)

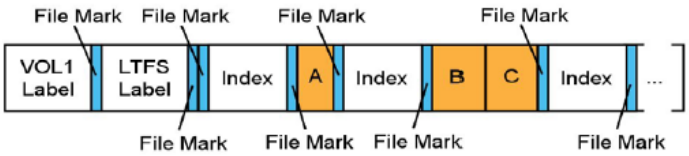
Data

- Set of sequential logical blocks

LTFS Label

```
<?xml version="1.0" encoding="UTF-8"?>
<ltfslabel version="2.3.0">
          <creator>IBM LTFS 1.2.0 - Linux - mkltfs</creator>
          <formattime>2013-02-01T18:35:47.866846222Z</formattime>
          <volumeuuid>30a91a08-daae-48d1-ae75-69804e61d2ea</volumeuuid>
          <location>
                    <partition>b</partition>
          </location>
          <partitions>
                    <index>a</index>
                    <data>b</data>
          </partitions>
          <blocksize>524288</blocksize>
          <compression>true</compression>
</ltfslabel>
```

LTFS Partition



LTFS Index

Index layout

- A generation number relative to other indexes in the volume
- A self pointer
  - Volume
  - Block position
  - Used to check Index validity
- A back pointer
  - Block position of the last index previously written in Data partition

LTFS Index

```
<?xml version="1.0" encoding="UTF-8"?>
<ltfsindex version="2.3.0">
 <creator>IBM LTFS 1.2.0 - Linux - ltfs</creator>
 <volumeuuid>30a91a08-daae-48d1-ae75-69804e61d2ea</volumeuuid>
 <generationnumber>3</generationnumber>
 <comment>A sample LTFS Index</comment>
 <updatetime>2013-01-28T19:39:57.245954278Z</updatetime>
 <location>
  <partition>a</partition>
  <startblock>6</startblock>
 </location>
 <previousgenerationlocation>
  <partition>b</partition>
  <startblock>20</startblock>
 </previousgenerationlocation>
 <allowpolicyupdate>true</allowpolicyupdate>
 <dataplacementpolicy>
  <indexpartitioncriteria>
   <size>1048576</size>
   <name>*.txt</name>
  </indexpartitioncriteria>
 </dataplacementpolicy>
 <volumelockstate>unlocked</volumelockstate>
 <highestfileuid>4</highestfileuid>
 <directory>
 ...
 </directory>
</ltfsindex>
```

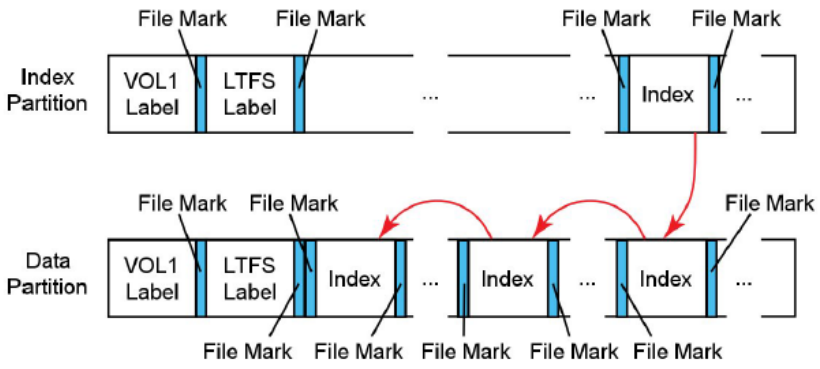
LTFS Index: Directory

```
<directory>
 <fileuid>1</fileuid>
 <name>LTFS Volume Name</name>
 <creationtime>2013-01-28T19:39:50.715656751Z</creationtime>
 …
 <contents>
  <directory>
   <fileuid>2</fileuid><name>directory1</name>
   <creationtime>2013-01-28T19:39:50.740812831Z</creationtime> <changetime>2013-01-28T19:39:56.238128620Z</changetime>
   <modifytime>2013-01-28T19:39:54.228983707Z</modifytime> <accesstime>2013-01-28T19:39:50.740812831Z</accesstime>
   <backuptime>2013-01-28T19:39:50.740812831Z</backuptime>
   <readonly>false</readonly>
   <contents>
    <directory>
     <fileuid>3</fileuid>
     <name>subdir1</name>
     <readonly>false</readonly>
     <creationtime>2013-01-28T19:39:54.228983707Z</creationtime>
     …
    </directory>
   </contents>
  </directory>
  <file>
   <fileuid>4</fileuid>
   <name>testfile.txt</name>
   <length>5</length>
   <creationtime>2013-01-28T19:39:51.744583047Z</creationtime>
   …
   <readonly>true</readonly>
   <extendedattributes></extendedattributes>
   <extentinfo>
    <extent>
     <partition>a</partition> <startblock>4</startblock> <byteoffset>0</byteoffset> <bytecount>5</bytecount> <fileoffset>0</fileoffset>
    </extent>
   </extentinfo>
  </file>
 </contents>
</directory>
```

LTFS Index: Extended Attributes

```
...
<directory>
 <fileuid>2</fileuid>
 <name>directory1</name>
 <creationtime>2013-01-28T19:39:50.740812831Z</creationtime>
 <changetime>2013-01-28T19:39:56.238128620Z</changetime>
 <modifytime>2013-01-28T19:39:54.228983707Z</modifytime>
 <accesstime>2013-01-28T19:39:50.740812831Z</accesstime>
 <backuptime>2013-01-28T19:39:50.740812831Z</backuptime>
 <extendedattributes>
  <xattr>
   <key>binary_xattr</key>
   <value type="base64">/42n2QaEWDSX+g==</value>
  </xattr>
  <xattr>
   <key>empty_xattr</key>
   <value/>
  </xattr>
  <xattr>
   <key>document_name</key>
   <value type="text">LTFS Format Specification</value>
  </xattr>
 </extendedattributes>
</directory>
...
```

LTFS Index Chain



WrapUp on FileSystems

- FS is a critical part of OS software stack
- VFS abstraction
  - Allows transparency and co-location
- FS design depend of device targeted
- Understanding FS concepts and implementation is mandatory for
  - Performance
  - Reliability
  - Security

Commissariat à l'énergie atomique et aux énergies alternatives
Centre DAM-Ile de France | 91297 Bruyères-le-Châtel Cedex
T. +33 (0)1 69 26 40 00 | F. +33 (0)1 69 26 70 86
Direction des applications militaires
Département sciences de la simulation et de l'information
Service informatique scientifique et réseaux
Etablissement public à caractère industriel et commercial | RCS Paris B 775 685 019