Linux File Systems

Adrien 'schischi' Schildknecht

July 18, 2014

Why FS matters

- Overhead: non-volatile memory is often the slowest component
- Reliability: no fault tolerated (data loss!)
- People expect more and more features...
  - Quotas
  - Snapshot
  - Versioning
  - Replication
  - Database features...

Section 1

Hardware

HDD

Flash

- ⊕ no moving heads nor rotating platters
- ⊕ good random access
- ⊕ low power
- ⊖ wear out (writing)
- ⊖ write in 2 phases
  - Clear group of pages, 128Ko+ (slow)
  - Write individual pages, 4Ko (fast)

Hardware Constraints

HDD:

- locality
- optimize seeking

SSD:

- TRIM

Common:

- Smallest unit is a block (reading/writing)
- Least possible writing

Section 2

Abstraction

File - Directory

File: store a named piece of data to later retrieve it.

- stream of bytes, let the programmer read raw bytes
- structural metadata: inode
- descriptive metadata: attributes (name, owner, . . .)

Directory: provide a way to organize multiple files

- hierarchies: directories can contain directories
- data structure which contains names and handles

- **FS**: machinery to store/retrieve data
- **block**: smallest unit writable by a disk or fs
- **metadata**: info about a data, but not part of it
- **attribute**: couple name/value
- **superblock**: area where a fs stores its critical info
- **inode**: place to store the metadata of a file
- **dentry**: holds inode's relation, allows fs traversal

Simple FS

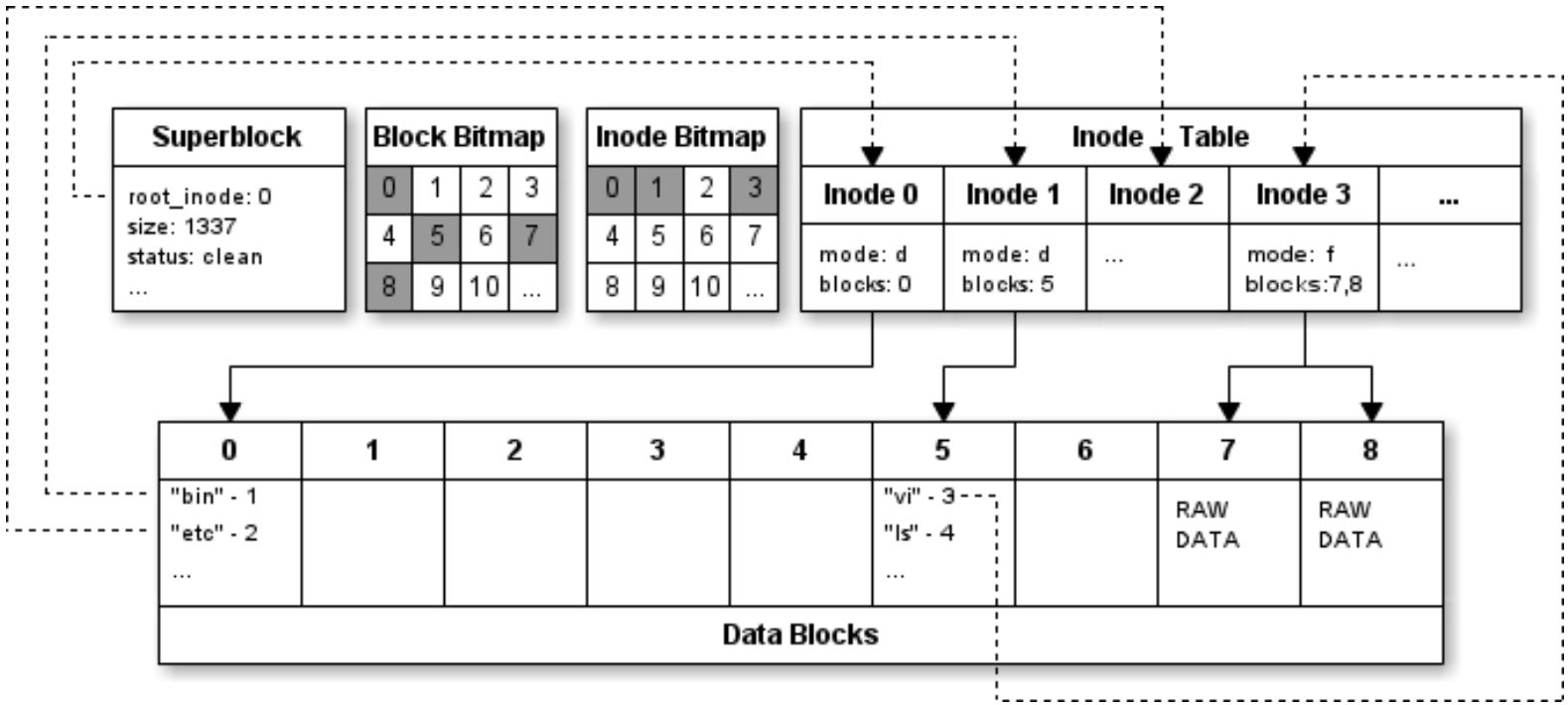
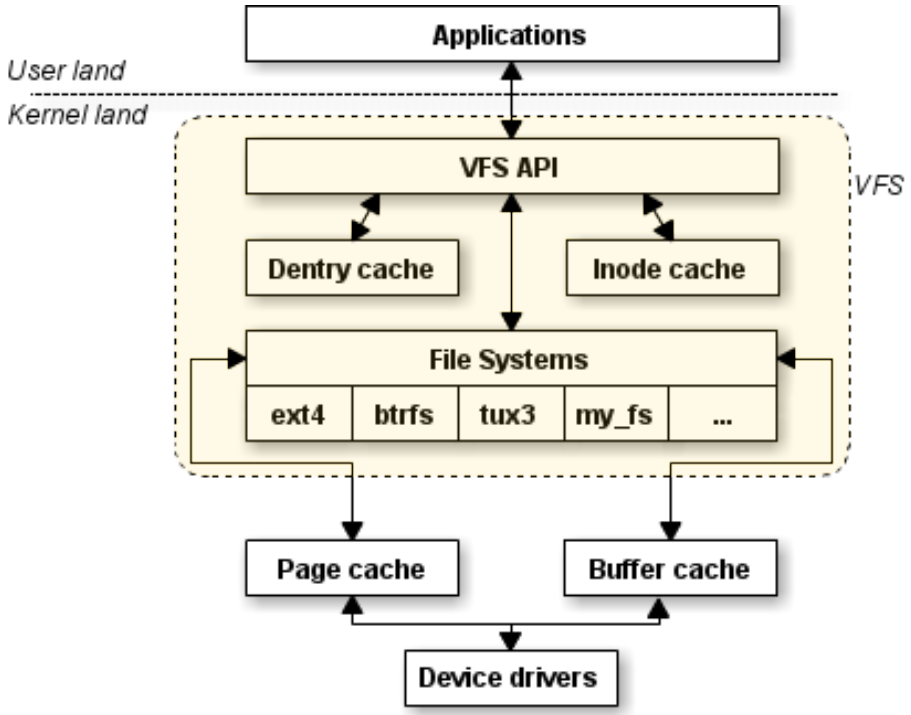


Figure: Relations between superblock, inodes, blocks, . . .

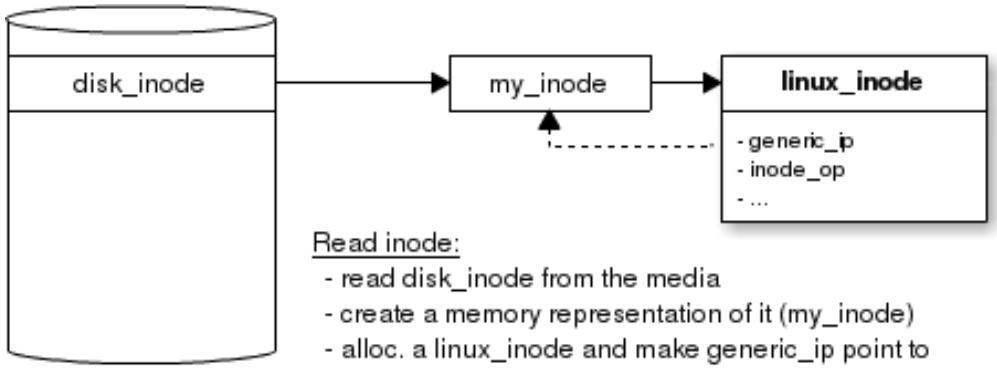
VFS: Why ?

- Keep track of available filesystems
- Provide an uniform interface
- Reasonable generic processing for common tasks
- Common I/O cache
  - Page cache
  - I-node cache
  - Buffer cache
  - Directory cache



The VFS API

- Linux defines generic function and structure but doesn't know anything about our fs
- Linux uses composition to store the fs structs
- Each struct contains a pointer to many member functions



Page cache

- Keep disk-backed pages in RAM
- Implemented with the paging memory managment
- It uses unused areas of memory.

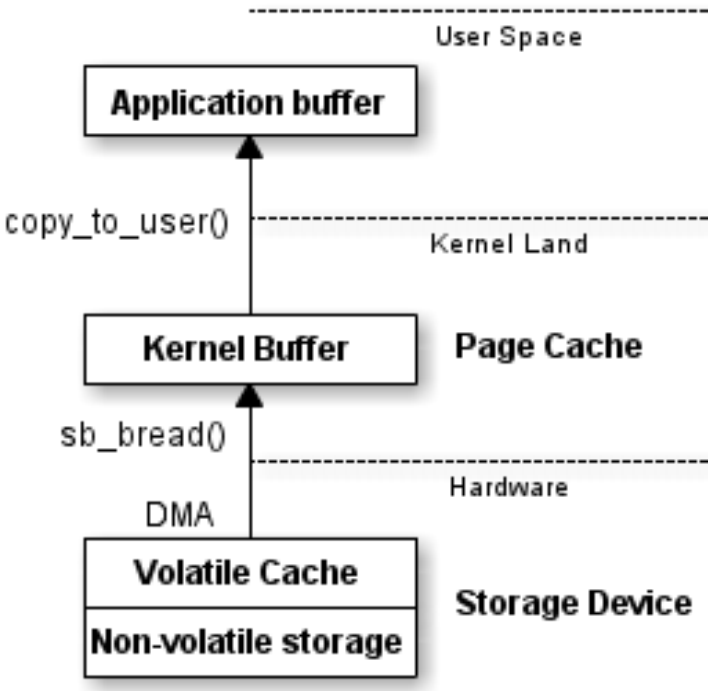
```
42sh> free -m
             total  used  free  shared  buffers  cached
Mem:          2947  2529   417     156      811     709
-/+ buffers/cache:  1007  1939
Swap:         1953     0  1953

```

Page cache

Reading:

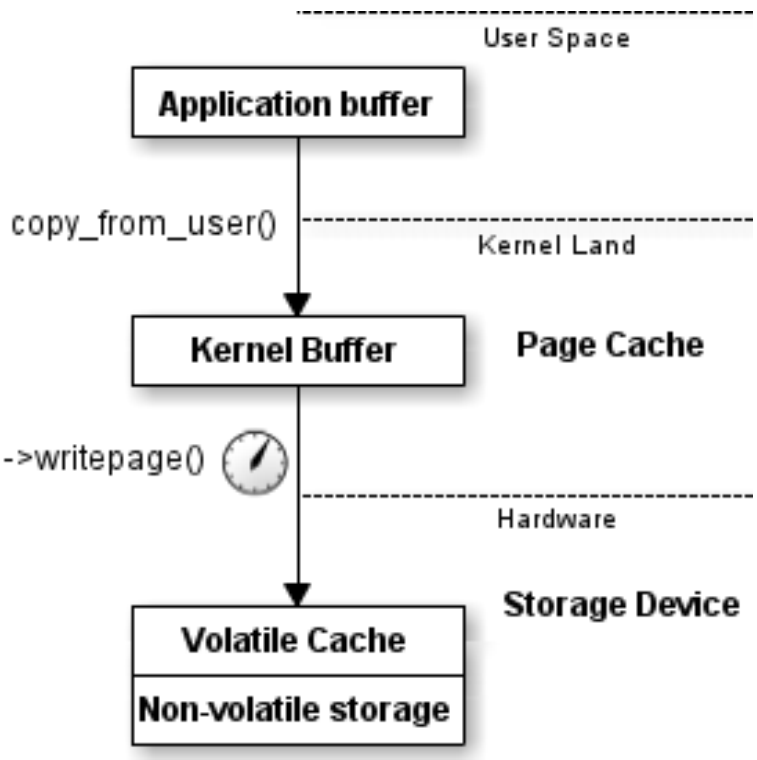
- read syscall
- if in the cache, retrieve it;
- otherwise read it from the device and add it to the cache



Page cache

Writing

- Copy buf to the page cache
- Mark the page as dirty
- The kernel periodically transfers all the dirty pages to the device



Writing

Why delay the write operations ?

- Temporal locality
- Seek optimization
- Group operations

You can bypass the cache by using **O_DIRECT**.

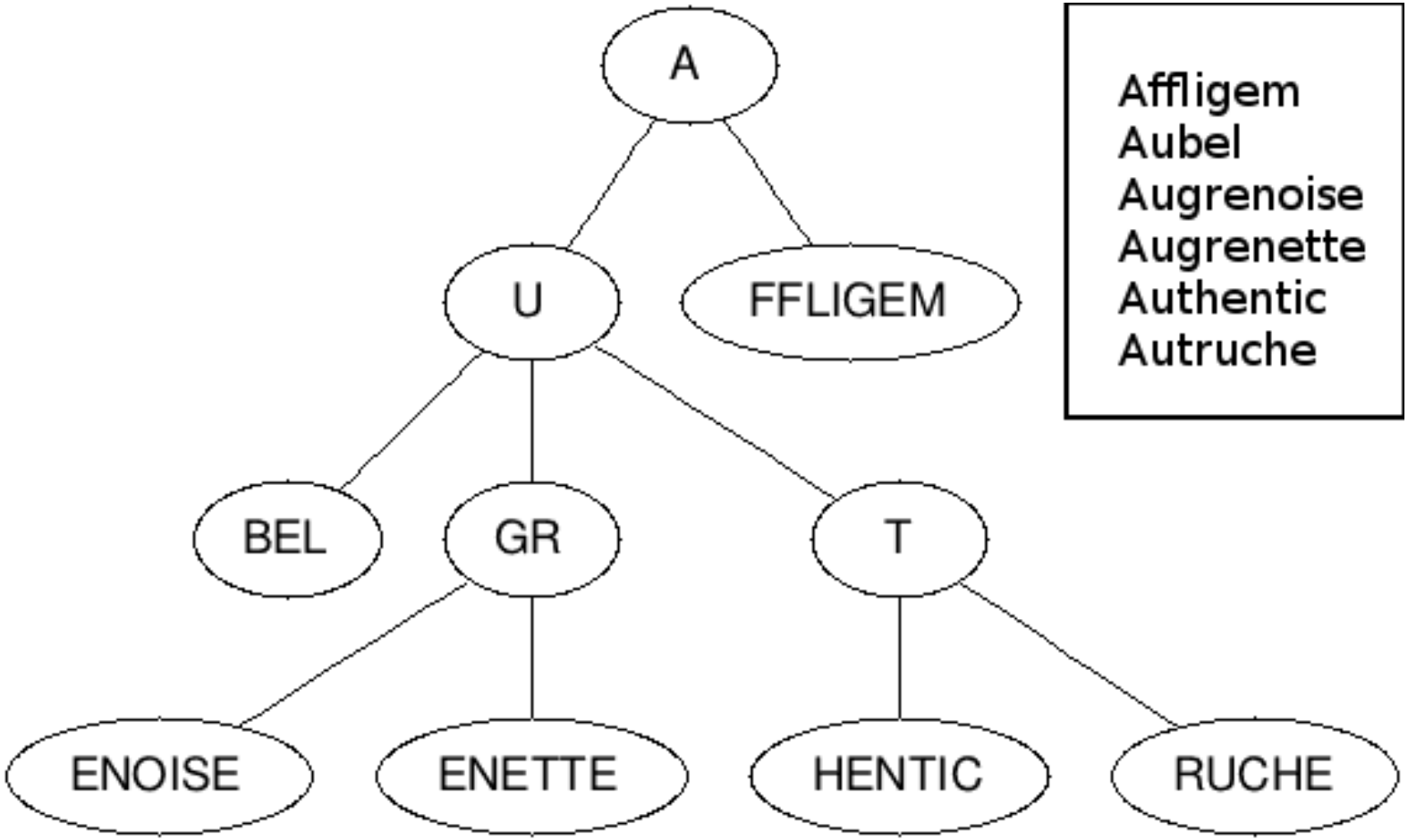
Writing

- When free memory shrinks below a specified threshold
- When dirty data grows older than a specific threshold
- Tunable parameters in /proc/sys/vm/
- You chan change the default I/O scheduler
- If more than a thresold percent of a process's adresse space is dirty, processes must wait for the I/O scheduler to flush the cache

```
1 cat /proc/sys/vm/dirty_expire_centisecs
2 3000
3 cat /proc/sys/vm/dirty_background_ratio
4 10
5 cat /proc/sys/vm/dirty_ratio
6 40
7
```

Radix Tree

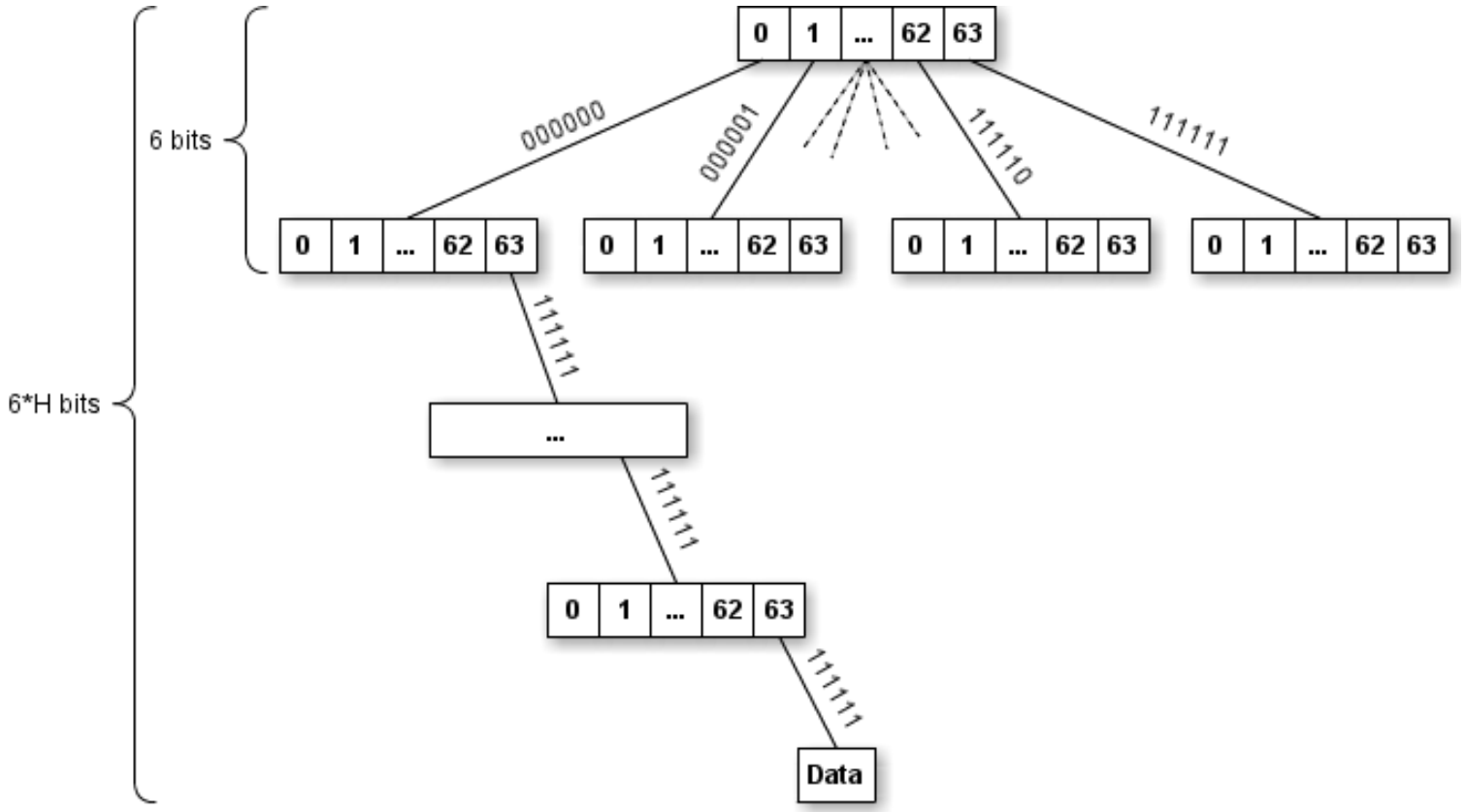
Radix Tree: a compact prefix tree



Linux Radix Tree

Radix Tree: a compact prefix tree

- Wide and shallow
- Each node contain 64 slots
- Each level is a 6 bits prefix



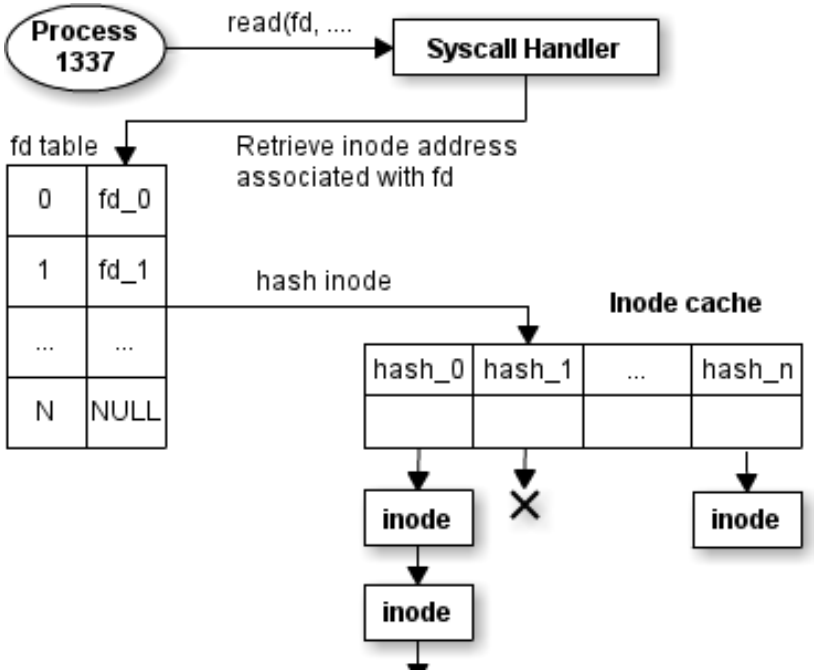
Linux Radix Tree

Additionnal feature: ability to associate tags with specific entries (to mark a page as dirty or under writeback for example) and retrieve them all easily.

I-node cache

Inode cache:

- Keep recently accessed file i-nodes
- The kernel retrieve the inode from the fd table of the application's address space
- Implemented as an open chain hash table, with blocks linked into a LRU lists
  - Used and dirty
  - Used and clean
  - Unused



Directory cache

d-cache: speed up accesses to commonly used directories

- Implemented as an open chained hash table, also linked into a LRU list
- Negative dentry for failed lookups
- Prehash with name, rehash with the dentry parent's address

Buffer cache

Block cache: interfaces with block devices, and caches recently used meta-data disk blocks. One LRU cache per-CPU.

- The array is sorted, newest buffer is at bhs[0]
- Discards the least recently used items first
- Implemented as an array of size 8 (caching 8 pages)

Section 3

Logging and Journaling

Without journaling

Removing a file :

- Remove its directory entry
- Mark the inode as free
- Mark data blocks as free

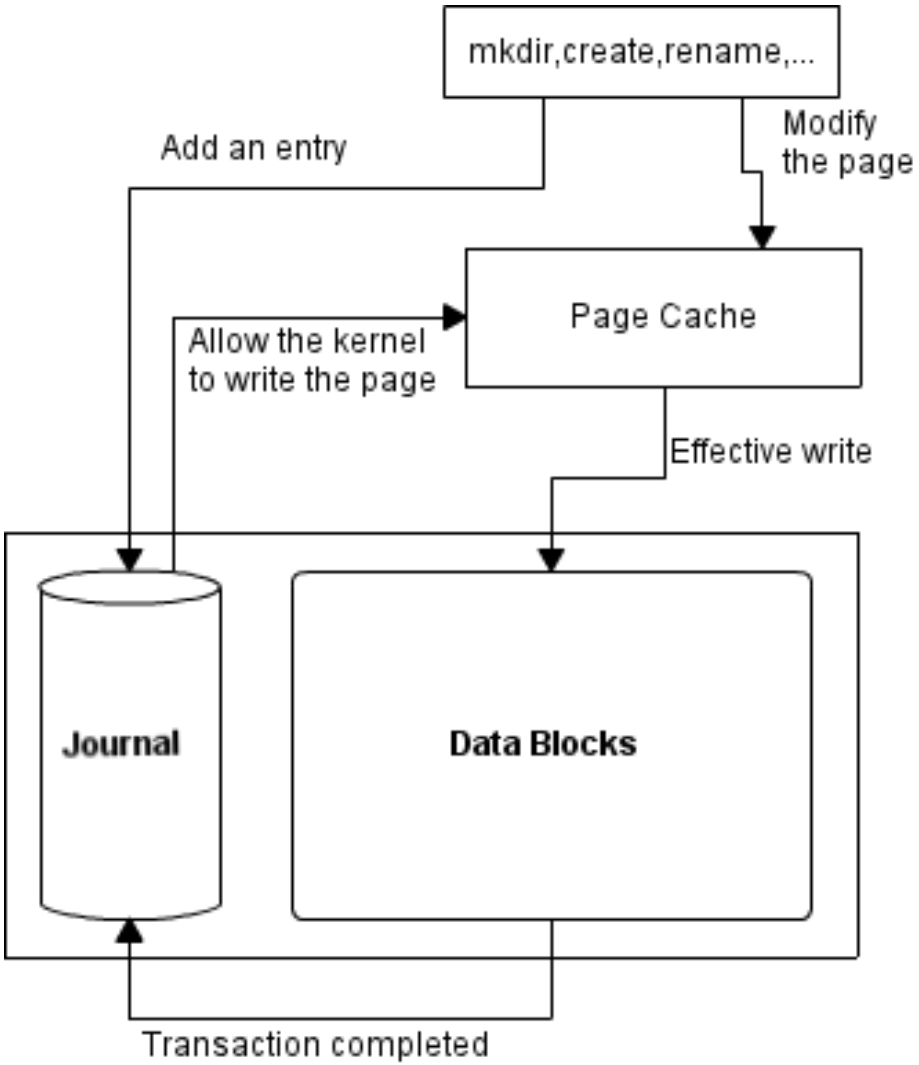
A crash between one of these steps leaves the fs in an inconsistent state, and thus needs to be fully checked (fsck)

Journaling

How to avoid partially written transactions ?

- **Transaction**: complete set of modifications made to the disk during one operation
- **Journal**: Fixed-size contiguous area on the disk (circular buffer)
- Writing to disk:
  - Add an entry to the journal
  - Allow the write to happen on disk
  - Mark the entry as completed
- If an entry is not completed when mounting, replay it

Journaling

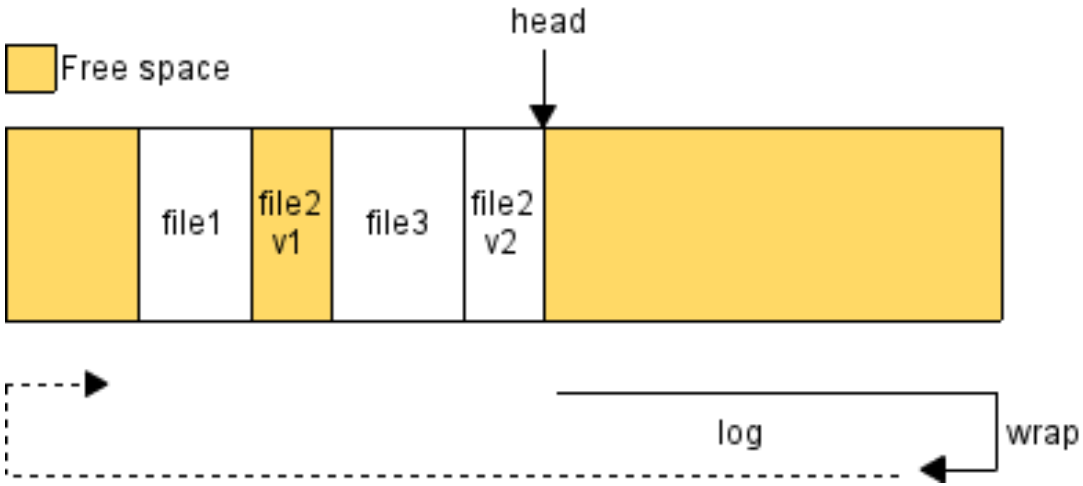

mkdir,create,rename,...
Add an entry
Modify the page
Page Cache
Allow the kernel to write the page
Effective write
Journal
Data Blocks
Transaction completed

Journaling

- ⊕ consistency of metadata
- ⊕ faster than fsck
- ⊖ data consistency is not ensured
- ⊖ redundancy of metadata writes

Logging

- The whole system data is structured in the form of a circular log
- Avoid writing data twice
- Copy-On-Write, mark the old verion as free and write at the end of the log



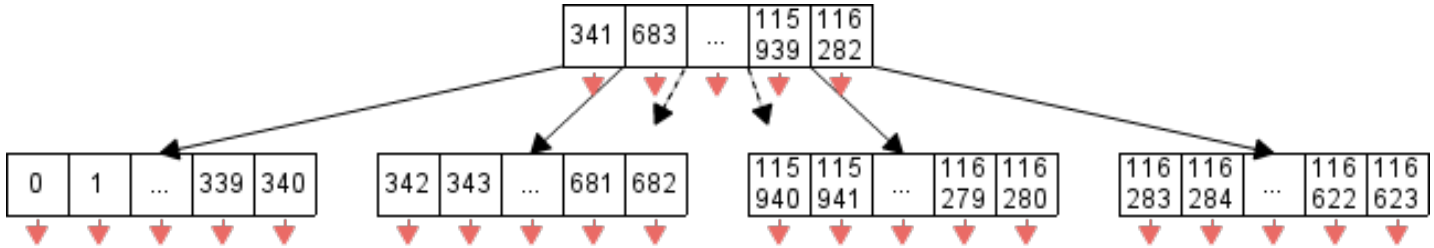
Logging

- ⊕ sequential writes
- ⊕ avoid redundant writes
- ⊖ slow random reads

Section 4

Real FS design

B-Tree



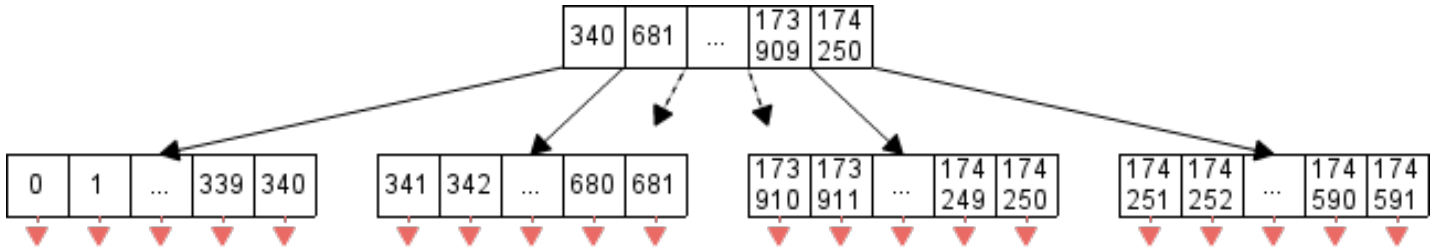

```c
struct btree_val {
    int key;
    void *data;
} typedef btree_val;
//sizeof(btree_val) = 8
```

```c
struct btree {
    btree_val values[N];
    btree *children[N+1];
} typedef btree;
//sizeof(btree)=8*N+(N+1)*4
```

$$4096 \geq N * 8 + (N + 1) * 4$$

$$N = 341$$

B+Tree




```c
struct bptree {
    int key[N];
    bptree *children[N+1];
} typedef bptree;
//sizeof(bptree)=4*N+(N+1)*4

```

```c
struct bptree_leaf {
    struct {
        int key;
        void *value;
        bptree_leaf *nxt; //opt
    } values[M];
} typedef btree_leaf;
//sizeof(btree)=12*N

```

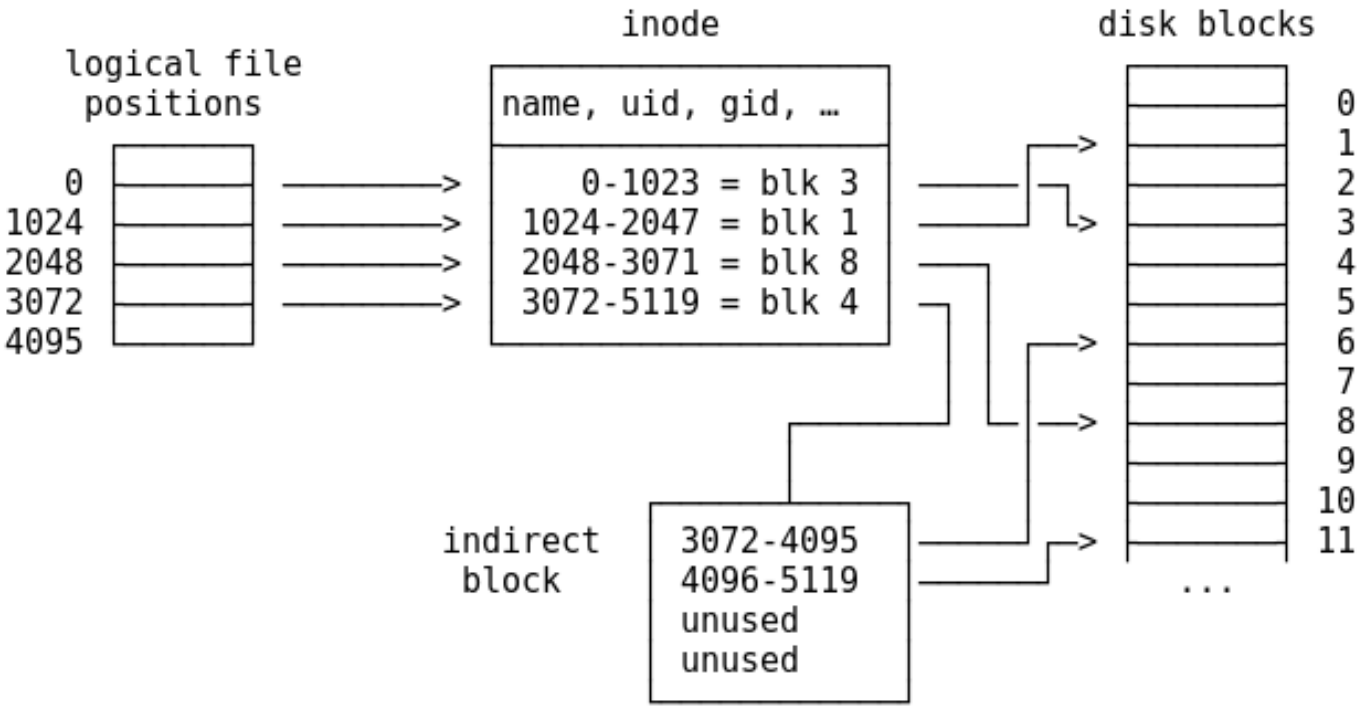
$$4096 \geq 4 * N + (N + 1) * 4$$

$$N = 511$$

$$4096 \geq 12 * M$$

$$M = 341$$

Block-Based allocation



Extents

- A chunk of blocks instead of a single block
- Still affected by fragmentation

```
1 struct ext3_extent {
2     __le32  ee_block;     /* first logical block extent covers */
3     __le16  ee_len;       /* number of blocks covered by extent */
4     __le16  ee_start_hi; /* high 16 bits of physical block */
5     __le32  ee_start;     /* low 32 bits of physical block */
6 };
7
```

Ext4

- Inode table
- Extents
- Journal
- Delayed block allocation
- Multi block allocator
- Online defragmentation
- Inline data
- Htree (a variant of B+tree)

Group 0 Padding	ext4 Super Block	Group Descriptors	Reserved GDT Blocks
1024 bytes	1 block	many blocks	many blocks

Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1 block	1 block	many blocks	many more blocks

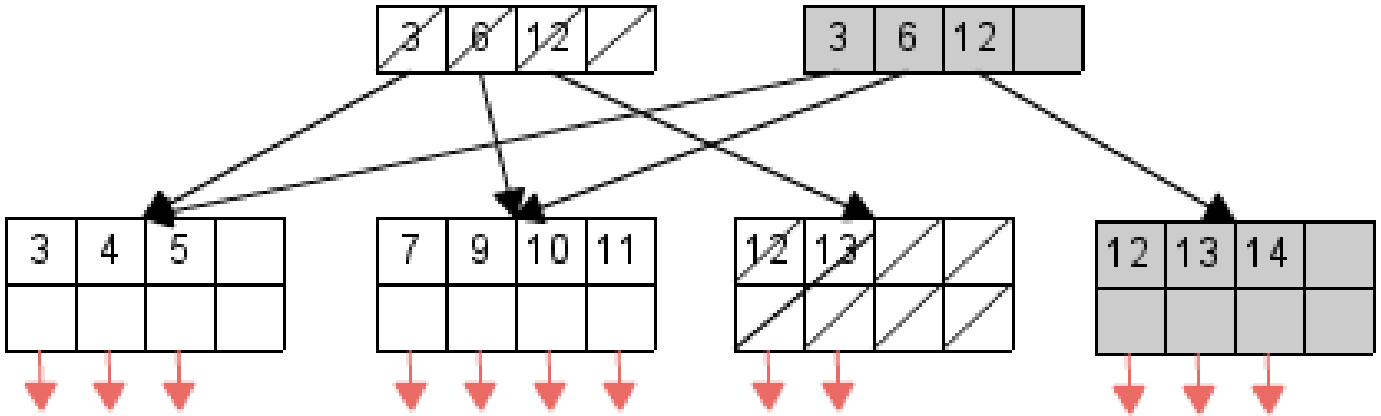
Btrfs

- Basically same features as ext4 (extents, inlining, . . .)
- Copy On Write metadata and data
- Transparent compression

Btrfs Tree

A B+tree containing a generic key/value pair storage.

- The same btree is used for all metadata



Section 5

Conclusion

Conclusion

```c
#define MEGA(S) ((S) * 1024 * 1024)

int main(int argc, char *argv[]) {
    char buf[4096];
    int fd = open("/home/schischi/foo", O_CREAT | O_WRONLY, 0660);

    if (argc == 2 && !strcmp(argv[1], "-f"))
        if (fallocate(fd, 0, 0, MEGA(700)) != 0)
            return 1;
    for (int i = 0; i < MEGA(700) / sizeof (buf); ++i)
        write(fd, buf, 4096);
    write(fd, buf, MEGA(700) % sizeof (buf));

    unlink("/home/schischi/foo");
    return 0;
}

```

```
$ repeat 100; ./a.out
    ./a.out  0.01s user 1.46s system 18% cpu 8.018 total

$ repeat 100; ./a.out -f
    ./a.out -f  0.00s user 1.01s system 13% cpu 7.440 total

```

Conclusion

Questions ?

schischi@lse.epita.fr
schischi - irc.rezosup.org

References

- FS design
  - Book "Practical File System Design" by Dominic Giampaolo
- VFS
  - http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git
  - http://lwn.net/Kernel/Index/
- Journaling, logging
  - http://pages.cs.wisc.edu/~remzi/OSTEP/file-lfs.pdf
  - http://research.cs.wisc.edu/wind/Publications/sba-usenix05.pdf
- Ext4
  - https://ext4.wiki.kernel.org/index.php/Ext4_Design
  - http://www.ibm.com/developerworks/library/l-anatomy-ext4/
- Btrfs
  - http://video.linux.com/videos/chris-mason-btrfs-file-system
  - http://atrey.karlin.mff.cuni.cz/~jack/papers/lk2009-ext4-btrfs.pdf