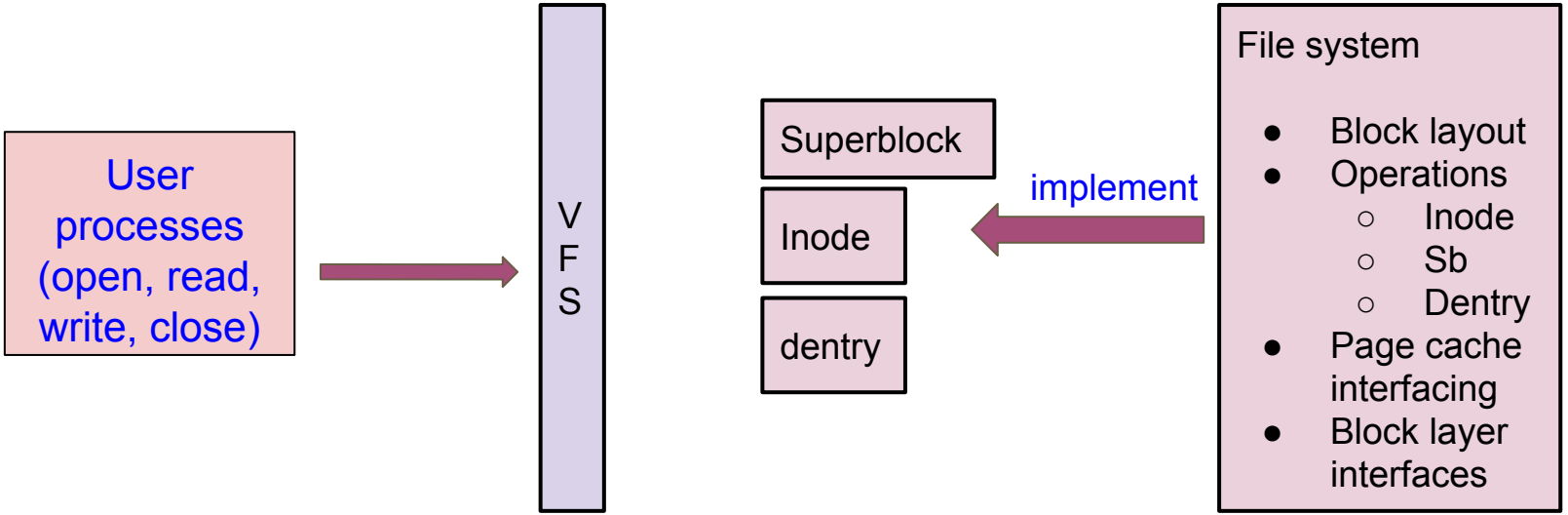
Case study: Linux extended filesystems

Recap


User processes (open, read, write, close)
VFS
Superblock
Inode
dentry
implement
File system
Block layout
Operations
Inode
Sb
Dentry
Page cache interfacing
Block layer interfaces

Let us design a filesystem

➔ Have you seen this picture?
➔ I wonder how ...

Let us design a filesystem

- Inodes, inodes and inodes...
  - Static allocation
  - Dynamic allocation
  - Where is my root inode?
- Find a free inode?
- How to find free blocks?
- How to walk through directories?

Inode table: static allocation vs dynamic allocation

Static allocation

Example: Inode size = ***I bytes***

Space reserved for N inodes = ***N * I bytes***

Dynamic allocation

Example: Inode size = ***I bytes***

If the FS has N inodes, used size = ***(N * I + X) bytes***, X is store index into inode

➔ Overhead of
- Finding an inode
- Allocating a new inode
- Freeing an inode

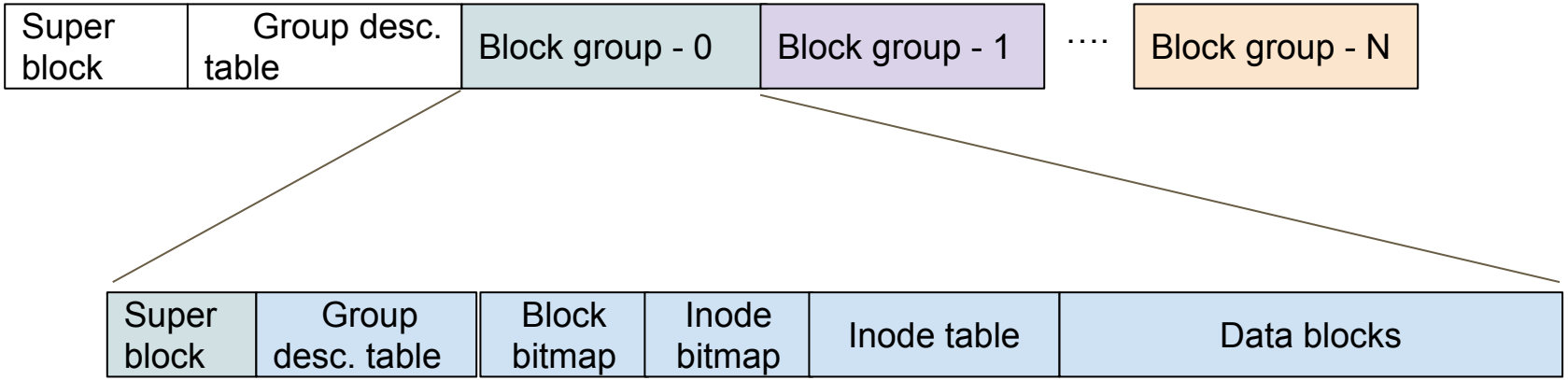
Inode table: finding a tradeoff

- Static allocation of inodes
  - Space wastage
  - Dynamic scalability
  - May lead to a lot of random I/Os

What could be the solution? OR a partial solution?

- Assumption: Maximum #of files supported file system has a (large) limit, but space used for inode tables $\propto$ no of used inodes
- Create more than one inode table (in different block groups)
  - Allocate related files in the same group

Block groups

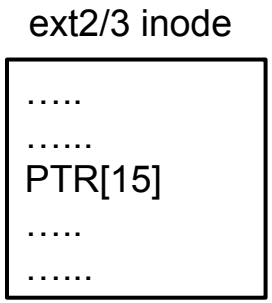


- ➔ If inode bitmap is one block, how many inodes?
  - ◆ How inode is unique?
- ➔ Should file data blocks span across groups?
- ➔ Why superblock and block desc repeated?

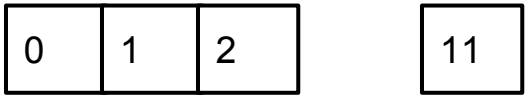
Illustration: operations

- Read inode (inode#)
  - inode# → Block group descriptor → Inode table → inode
  - BG = (inode - 1) / sb.inodes_per_blockgroup
- /home/user/$ grep sqrt *.c
  - Assume inode for “user” is known
  - What all operations needed?
- /home/user/$ touch newfile
  - Assume inode for “user” is known
  - Operations?

From inode to data blocks

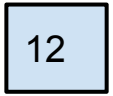


Direct {PTR [0] to PTR [11]}



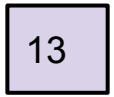
File block address (0 -11)

SI {PTR [12]}



File block address (12 -1035)

DI {PTR [13]}



File block address (1036 to 1049611)

DI {PTR [14]}

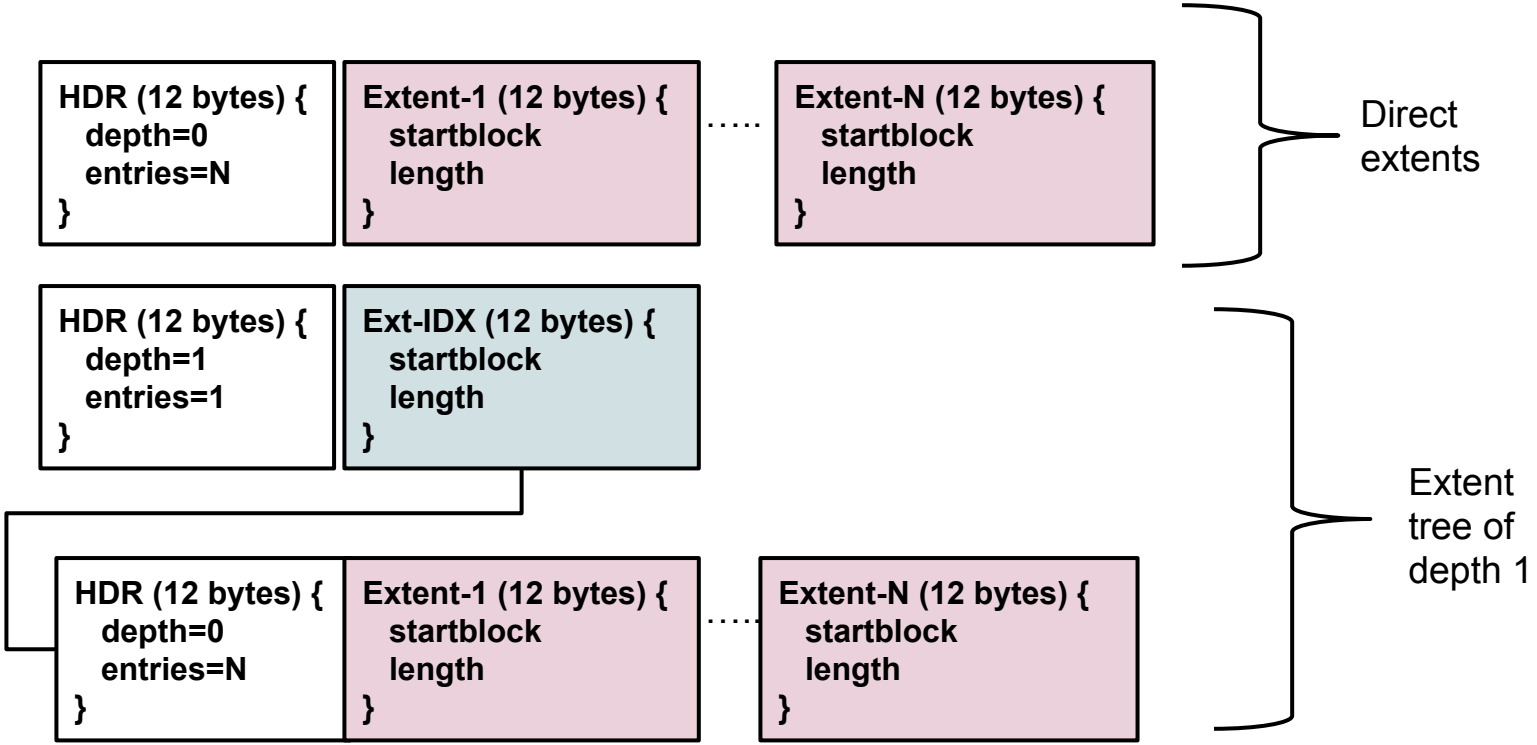
File block address (?? to ??)

All is not well with indexed organization !

- ➔ Fast access for small sized files
- ➔ Decent file size scalability
- ➔ But ....

- ➔ For a file size of 200 KB
  - ◆ One single indirect index is needed
- ➔ Why not use {block#, length}?

Ext4 extents and extent tree



Filesystem: crash consistency, recovery

- ➔ Multiple I/O operations (writes) required for many operations
  - ◆ Atomicity guarantee @ a sector level operation

- ➔ Example scenarios:
  - ◆ Append to a file → (1) update block PTR index/extent from inode (2) mark block used in block bitmap. Crash between 1 and 2 → same block used twice!
  - ◆ Create a file → (1) allocate inode (2) create an entry in directory data block. Crash between 1 & 2→ inode with no parent !

- ➔ Filesystem consistency check ...

Sanity check: fsck

- ➔ During FS mount, check if it had been cleanly unmounted when it was last used
  - How to know?
- ➔ Perform a walk from the FS root
  - Cross check meta-data (bitmaps, inode table) consistency
  - Reverse reachability checks

Journals (>= ext3)

- ➔ Remember Redo-log and Undo-log concepts of databases?
- ➔ Similar idea, redo-log used by ext3
    1. Log before operation
    2. Perform disk operations
    3. Mark ”success” after all operation complete
- ➔ Fsck can only redo operations for unsuccessful log entries
- ➔ Different modes of journalling
    a. Only metadata
    b. metadata and data etc.

Useful links

https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout

www.nongnu.org/ext2-doc/ext2.html

http://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf