On The Way to a Healthy Btrfs Towards Enterprise

Miao Xie

<miaox@cn.fujitsu.com>

Tagged with “experimental”

- Oops still remains some.
- Fsck is at experimental stage.
- Immature for production use.

Our goal

- Improve btrfs to be fit for production use.

Btrfs that enterprise requests for

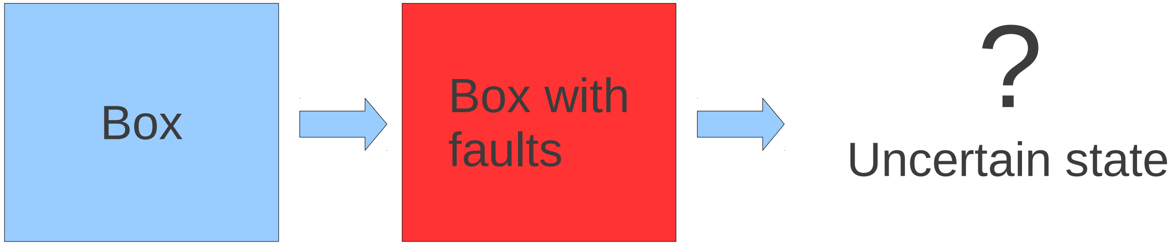
- Good performance
- Reliability
- Scalability
- Fault tolerance
- Features
  - Snapshot
  - Integrated multiple devices support
  - transparent compression
  - etc

Our progress

- Error handling infrastructure
- Free space cluster per node
- Snapshot aware defrag
- Inode cache
- Per file cow and compression control
- Extent buffer cache
- Rbtree lock contention
- A great amount of bug fixes and cleanups

Progress:
Error handling infrastructure

- Forced readonly mounts on errors
  - Being fault tolerant of disk corruptions
  - Build a framework which can flip btrfs to readonly when there are errors
  - Replace BUG() and BUG_ON()



Box

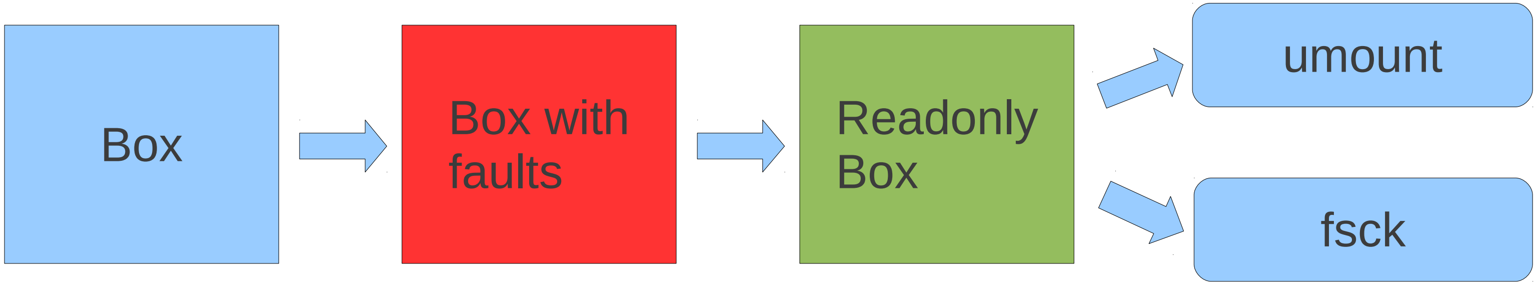
Box with faults

?

Uncertain state

w/o patch

w patch



Box

Box with faults

Readonly Box

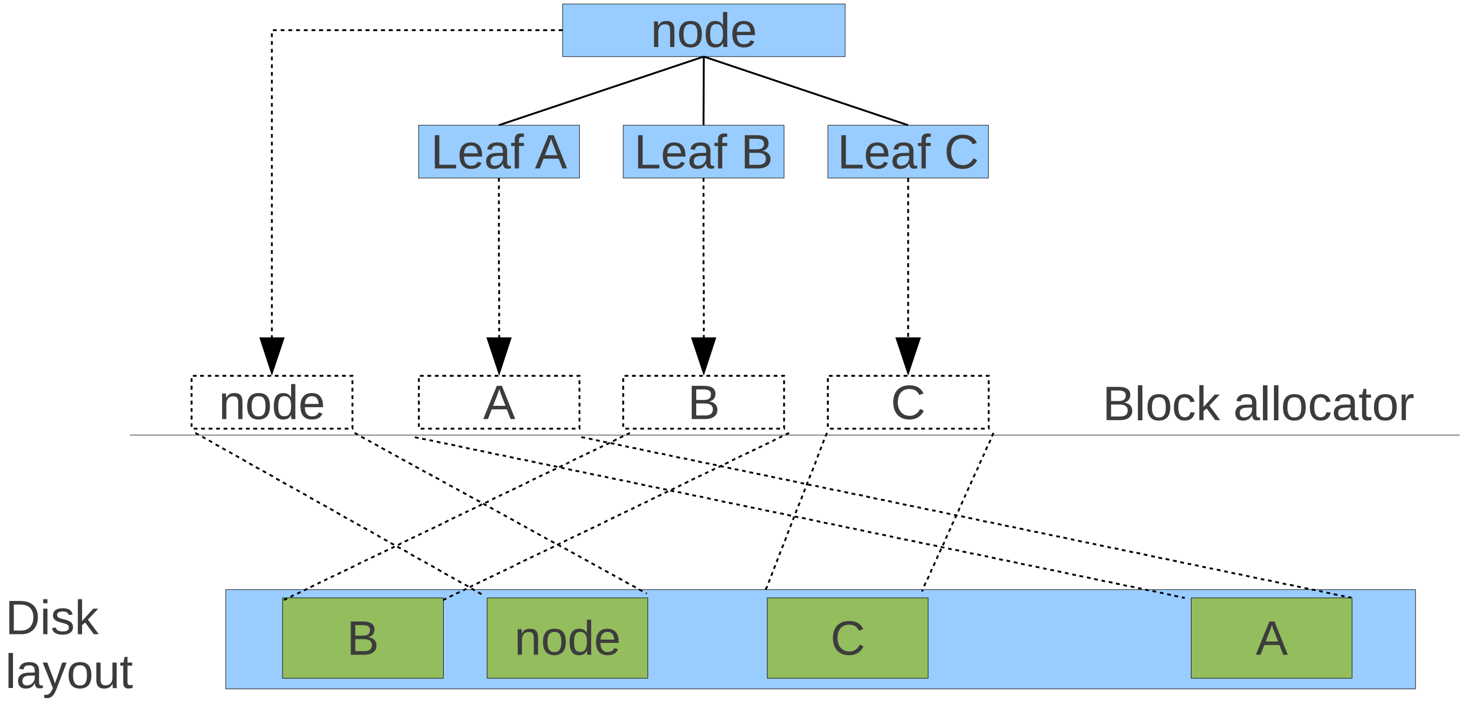
umount

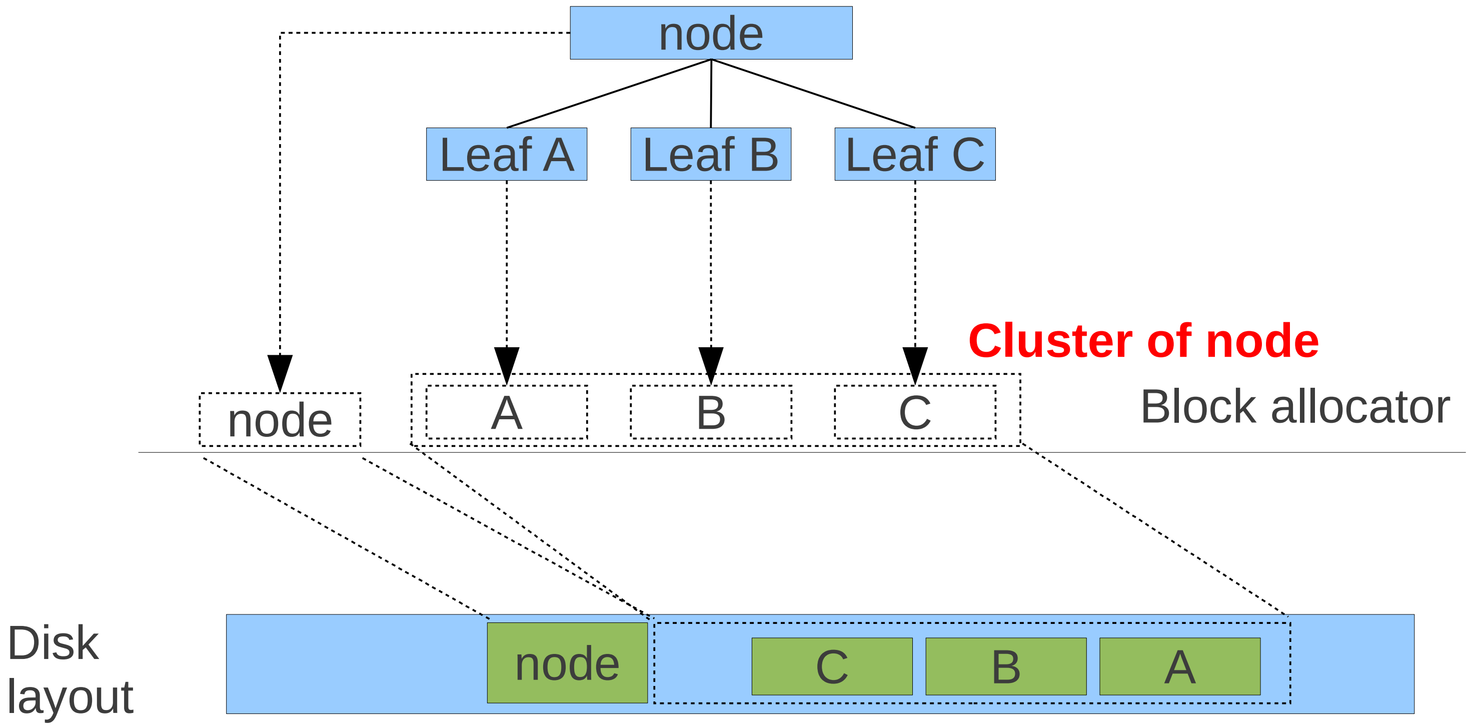
fsck

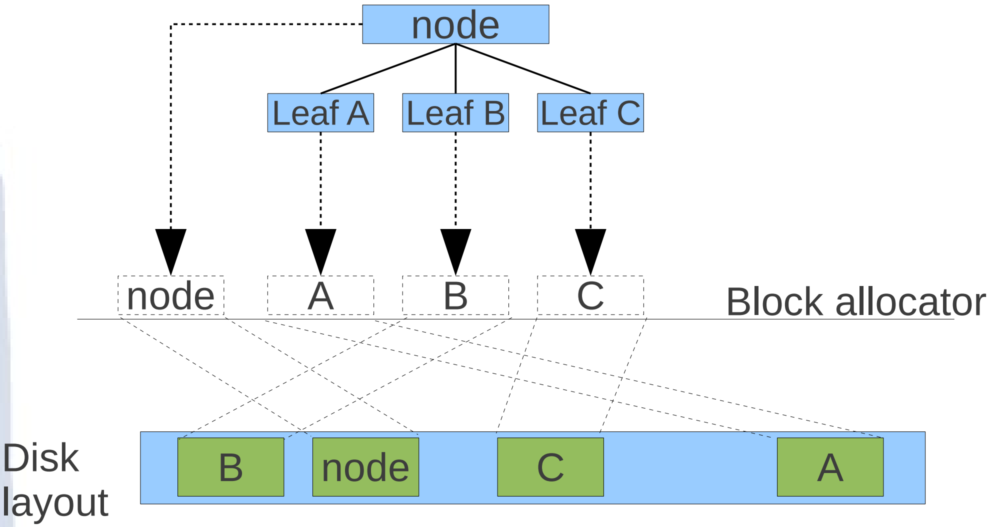
Progress:
free space cluster per node

- Reduce metadata fragments
- Improve (small files) sequencial read performance

WAFL(Write Anywhere File Layout)




node
Leaf A
Leaf B
Leaf C
Cluster of node
node
A
B
C
Block allocator
Disk
layout
node
C
B
A



node

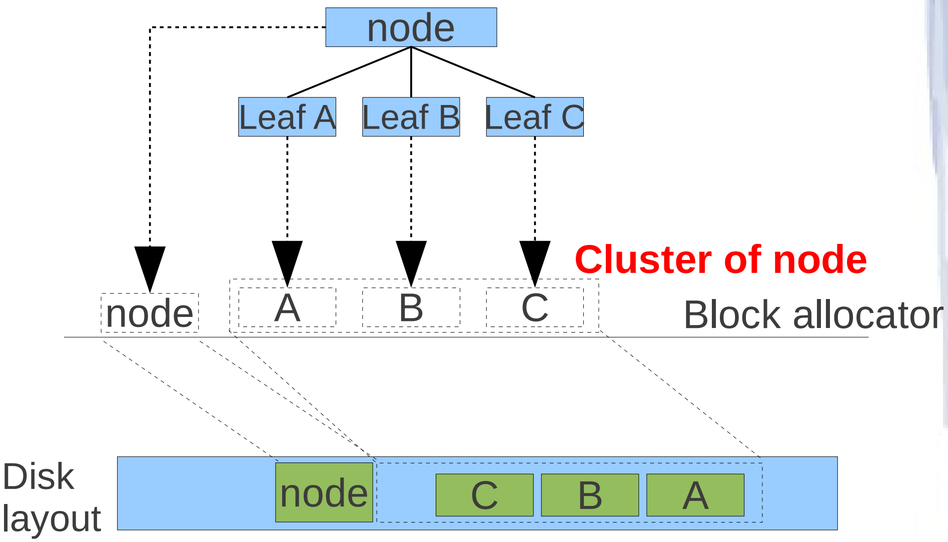
Leaf A Leaf B Leaf C

node A B C

Block allocator

Disk layout

B node C A



node

Leaf A Leaf B Leaf C

Cluster of node

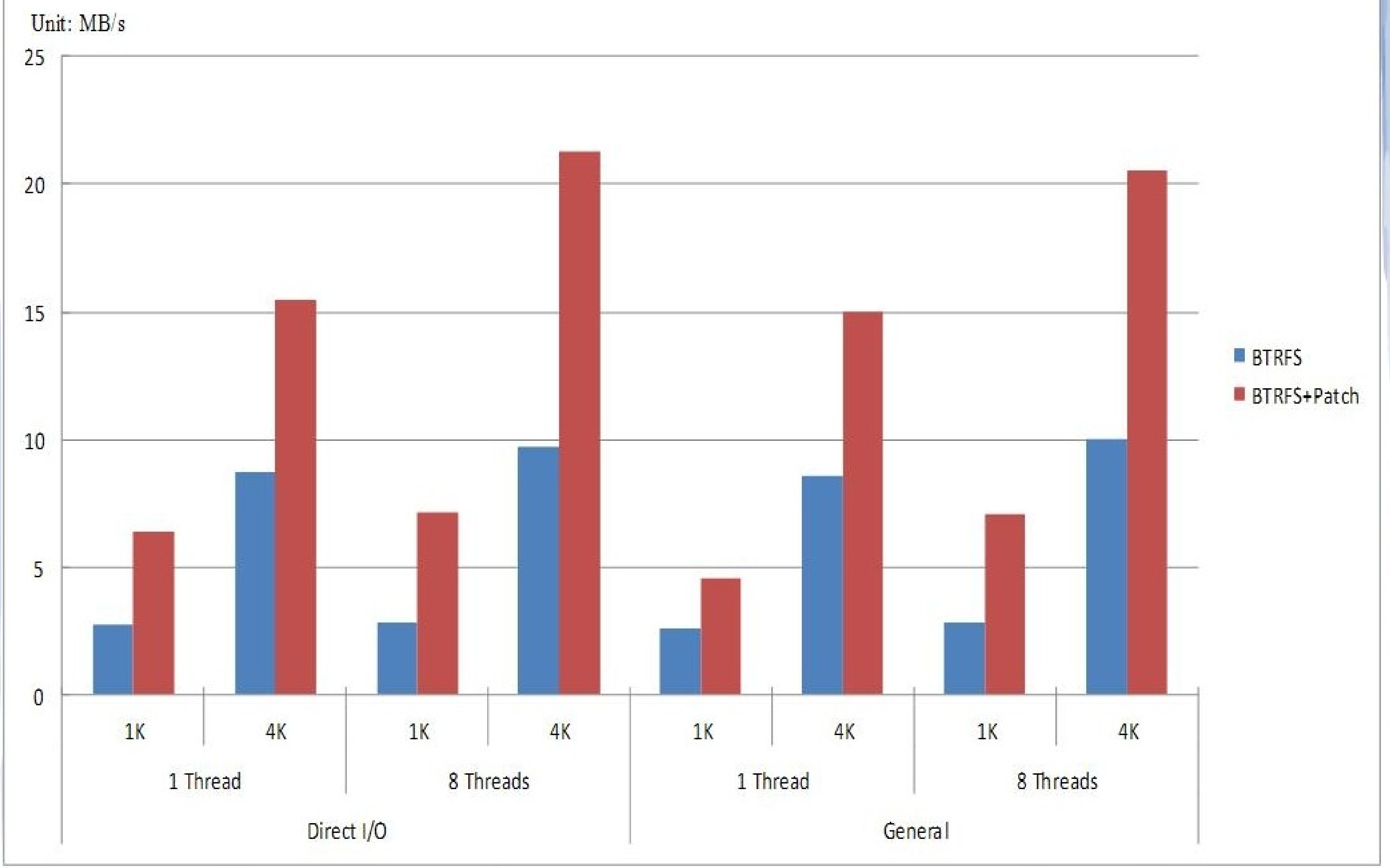
node A B C

Block allocator

Disk layout

node C B A

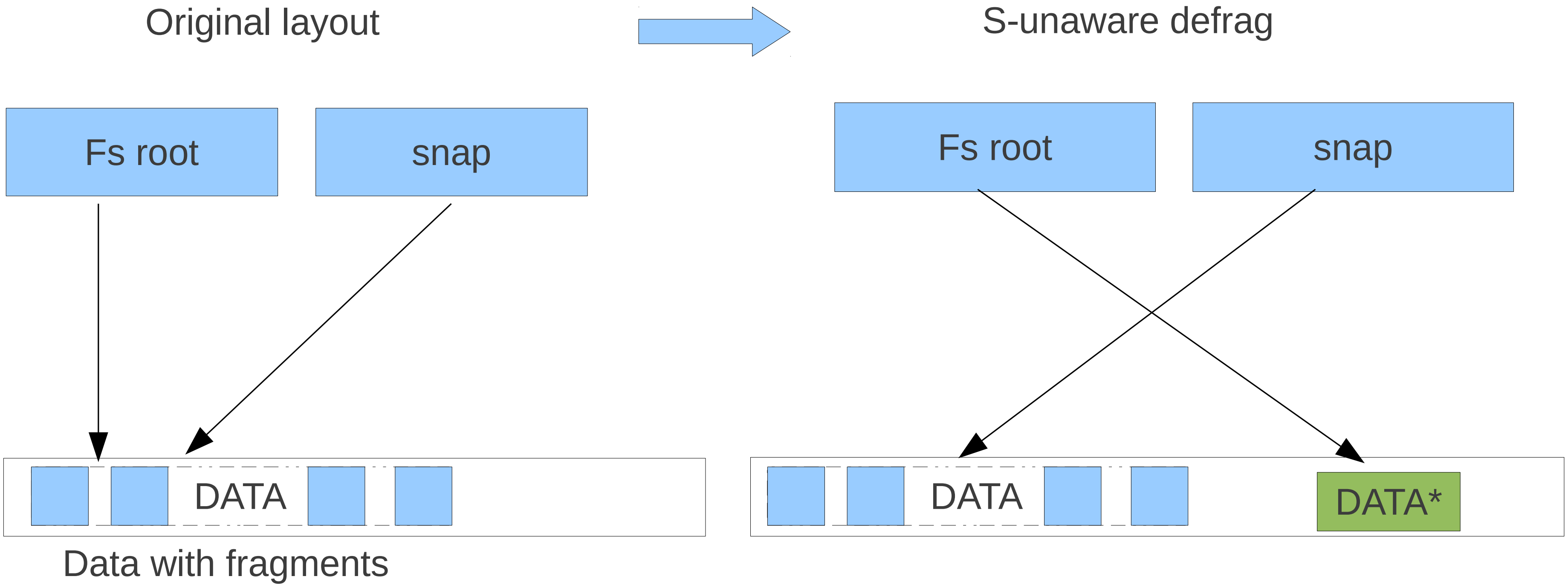
Small files sequential read (Based on 2.6.38)



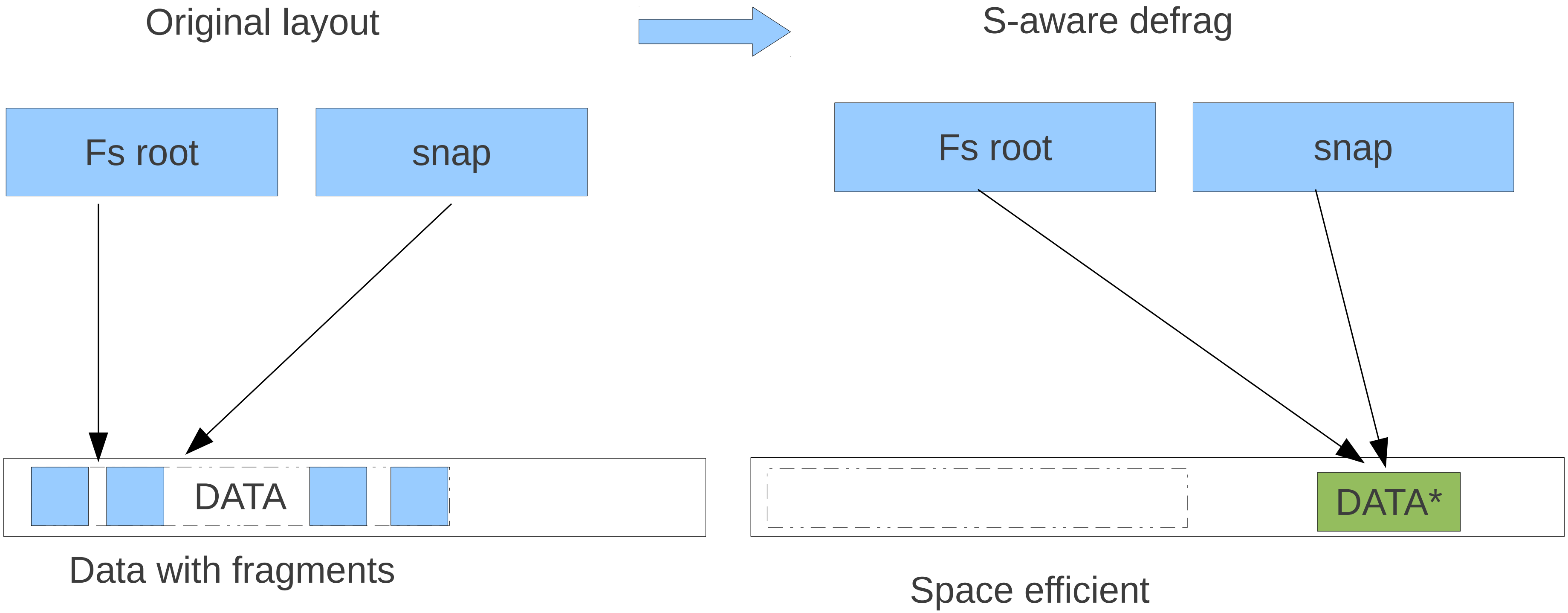
Progress:
snapshot aware defrag

- Make defragment code preserve the sharing among snapshots.
  - Btrfs has designed back references for this

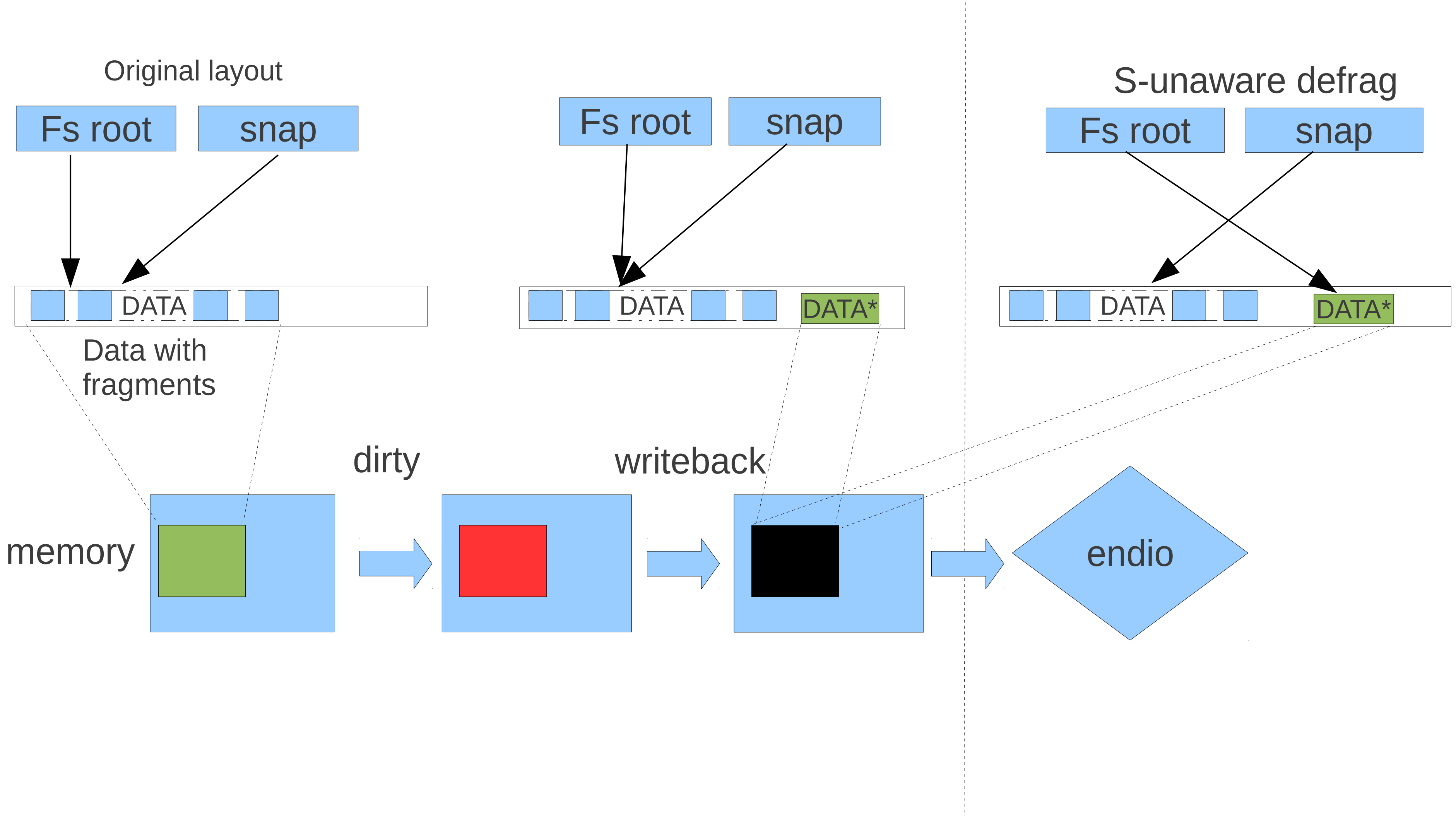
Snapshot unaware defragment

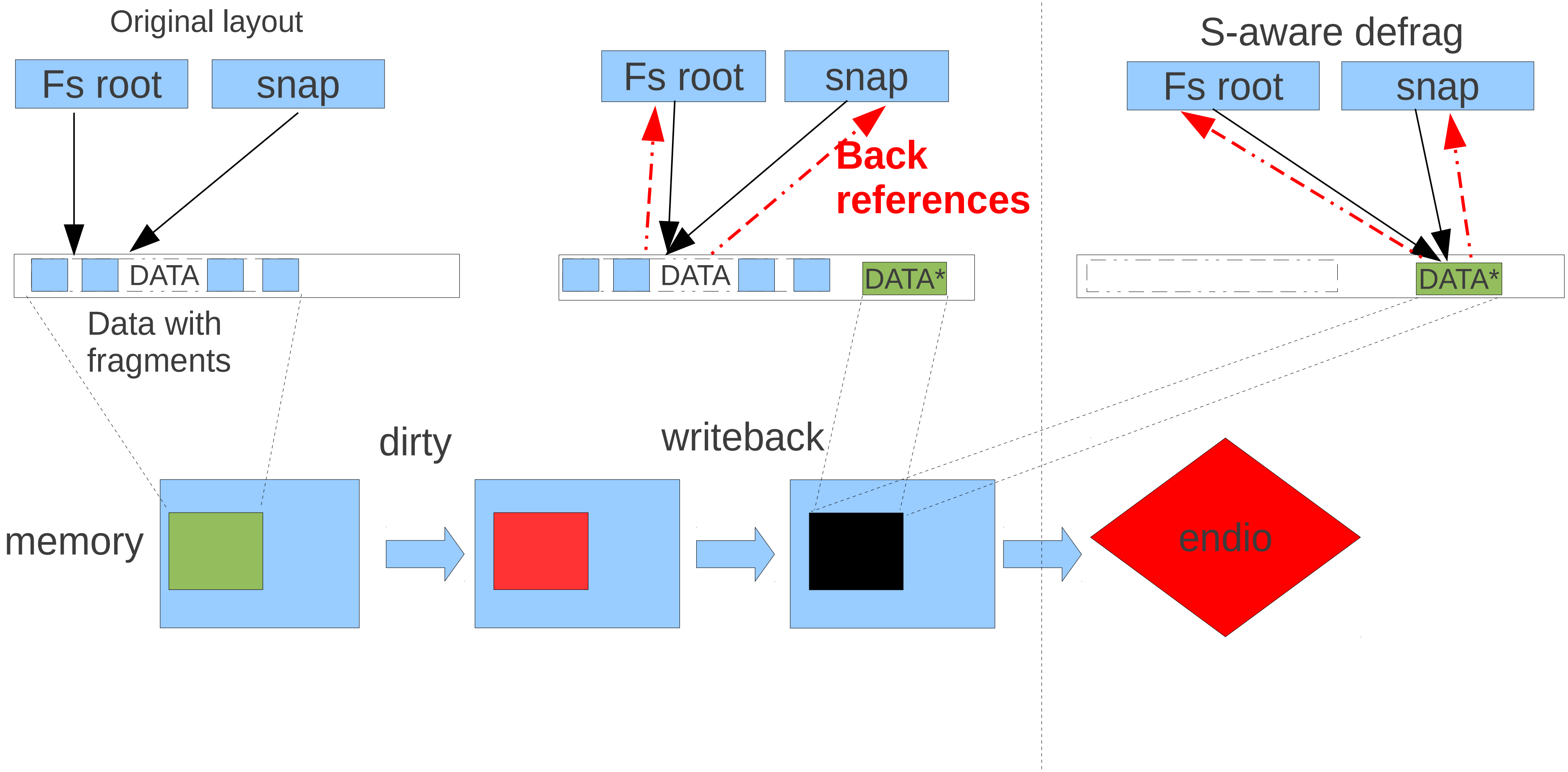


Snapshot aware defragment



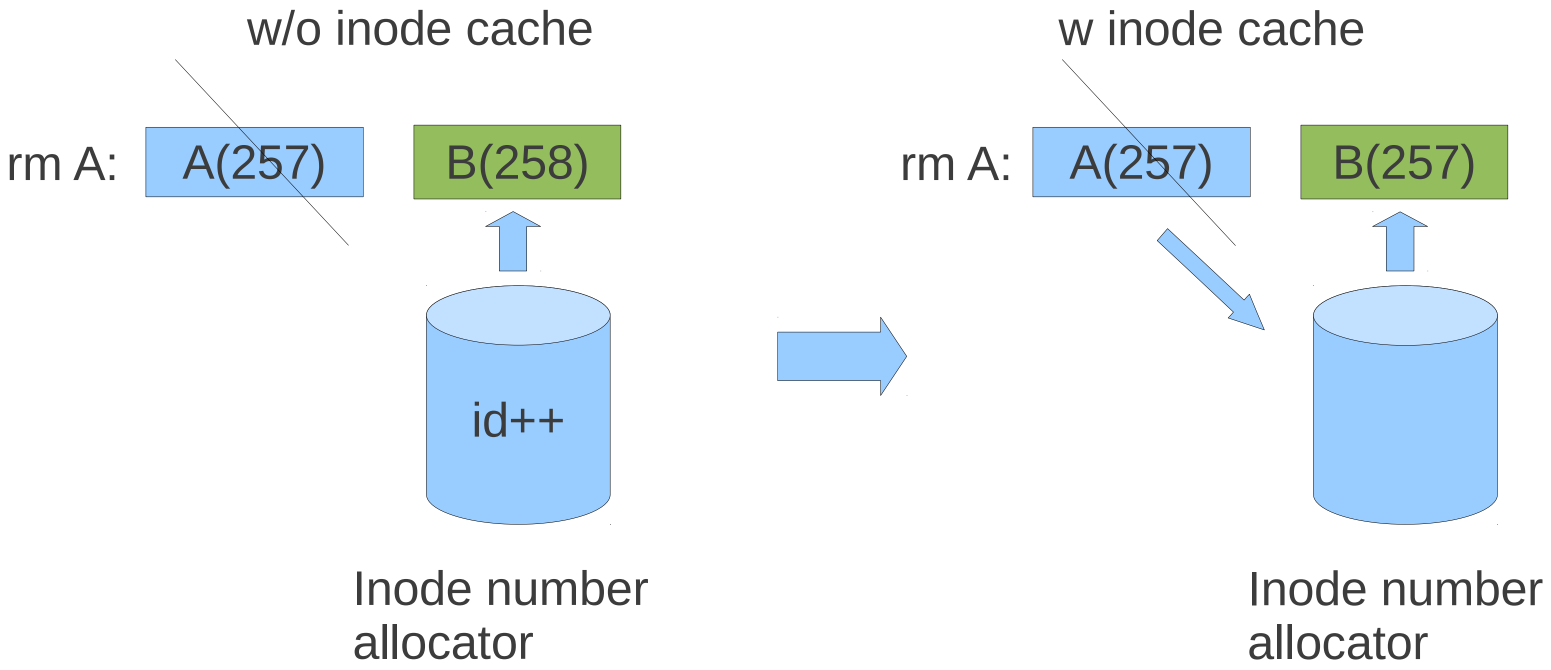
How does defragment work




Original layout
Fs root
snap
DATA
Data with fragments
Fs root
snap
DATA
DATA*
Back references
S-aware defrag
Fs root
snap
DATA*
memory
dirty
writeback
endio

Progress:
inode cache

- Without inode cache,
  - Will not reclaim inode number when deleting files
  - It will not reuse inode number

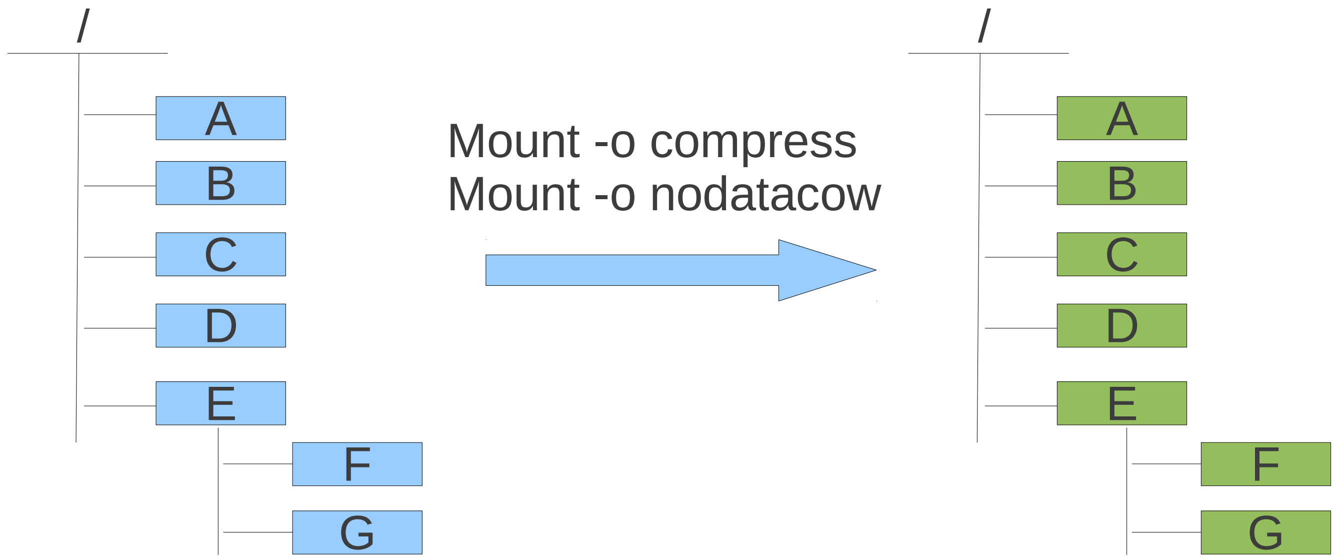


w/o inode cache

w inode cache

Progress:
Per file cow and compression control

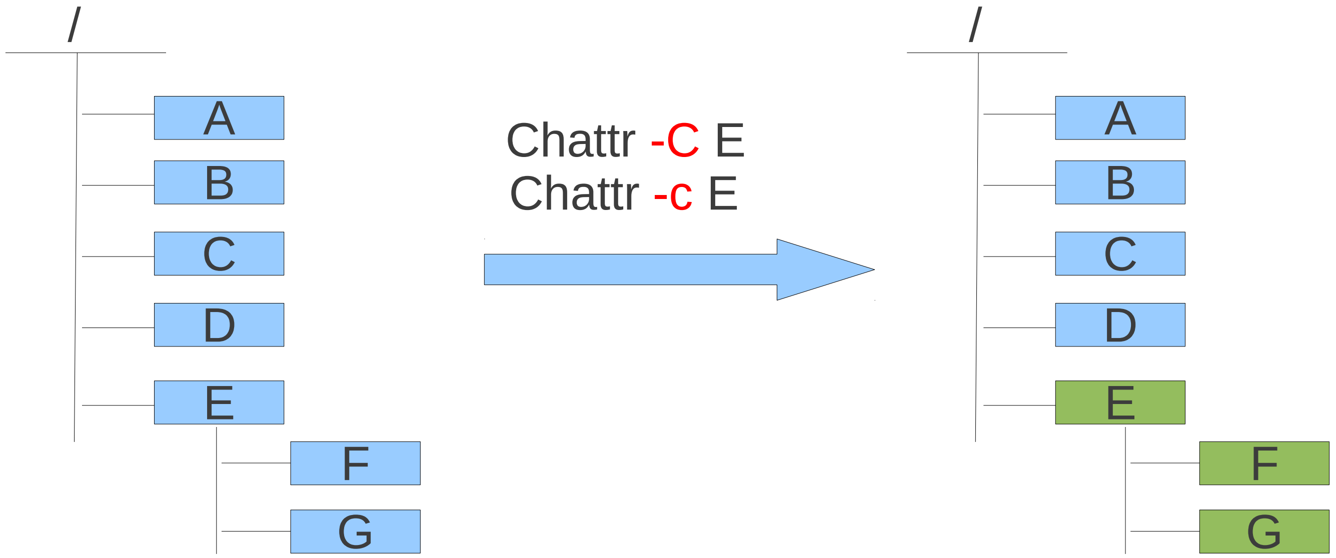
- Beside mount options, we need to control these flags on a per-inode basis.



Mount -o compress
Mount -o nodatacow

w/o patch

w patch

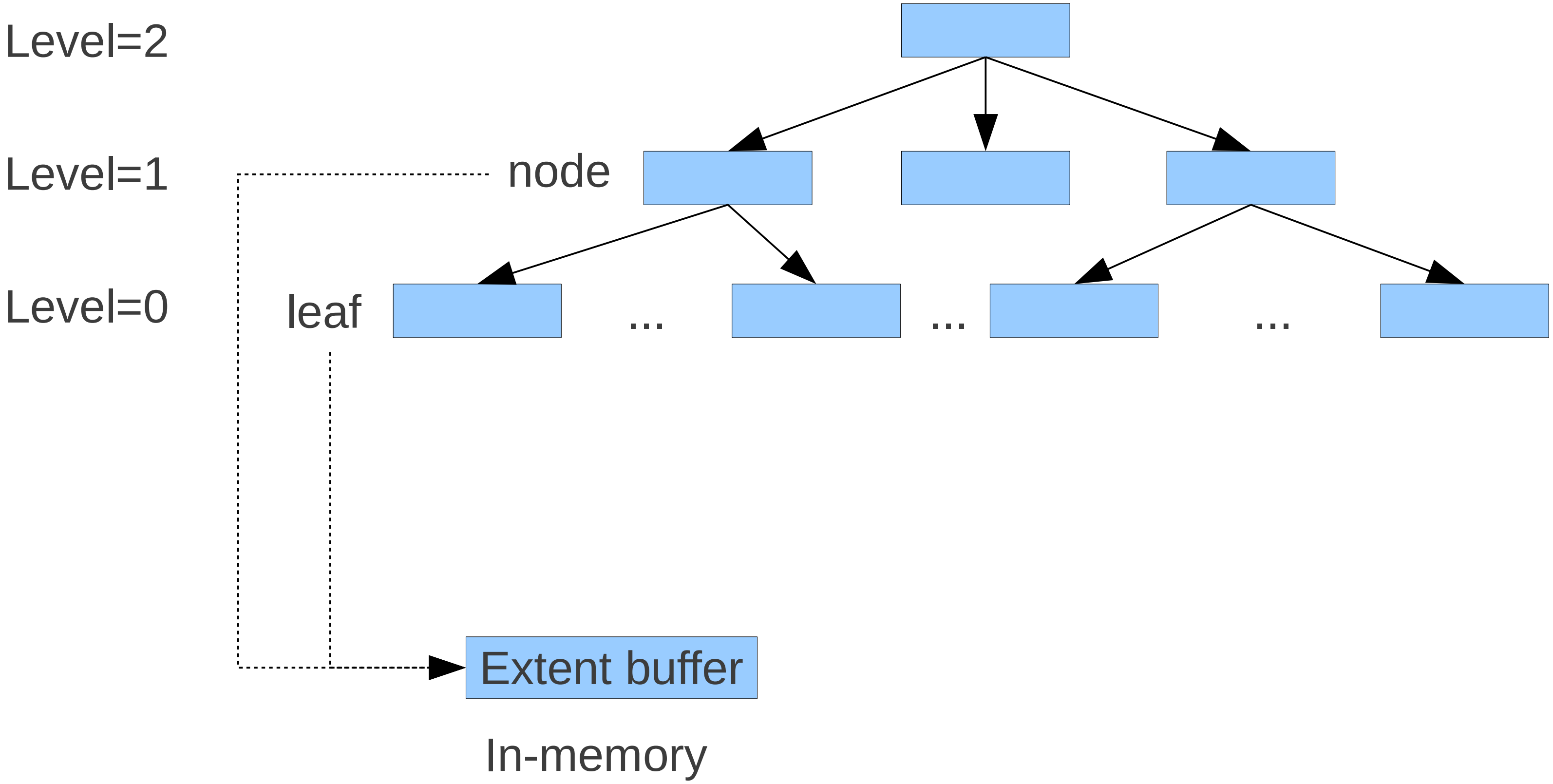


Chattr -C E
Chattr -c E

Progress:
extent buffer cache

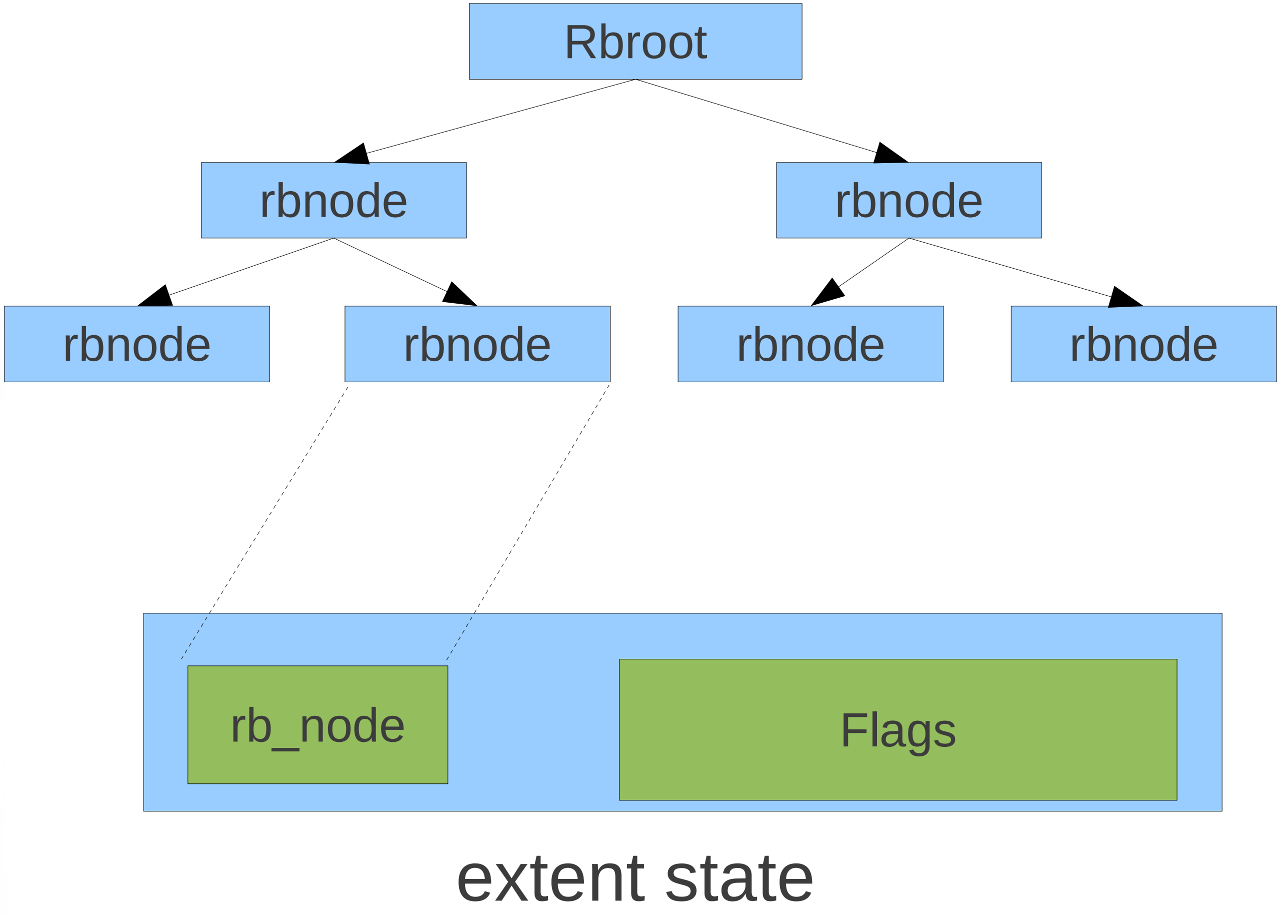
- Extent buffer is a basic unit of metadata
- Expensive on searching and reading
- Cache misses depend on workloads

What is extent buffer

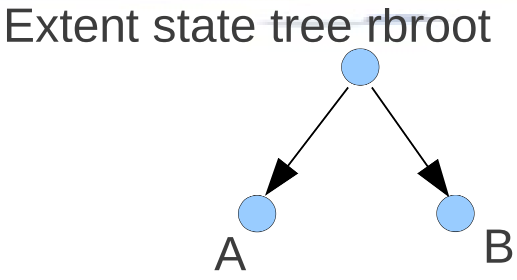


Progress: rbtree lock contention

- Why
  - Lock contention is really critical on performance
- How
  - Some rbtrees are domained by reads
  - Lockless read
- What
  - Build 'read mostly' circumstance
    - Find where the write locks are held
    - Try to reduce them as much as possible
  - Apply RCU, or rwlock



Flags:
- EXTENT_DIRTY,
- EXTENT_LOCKED,
- EXTENT_UPTODATE,
- EXTENT_DELALLOC,
- etc.

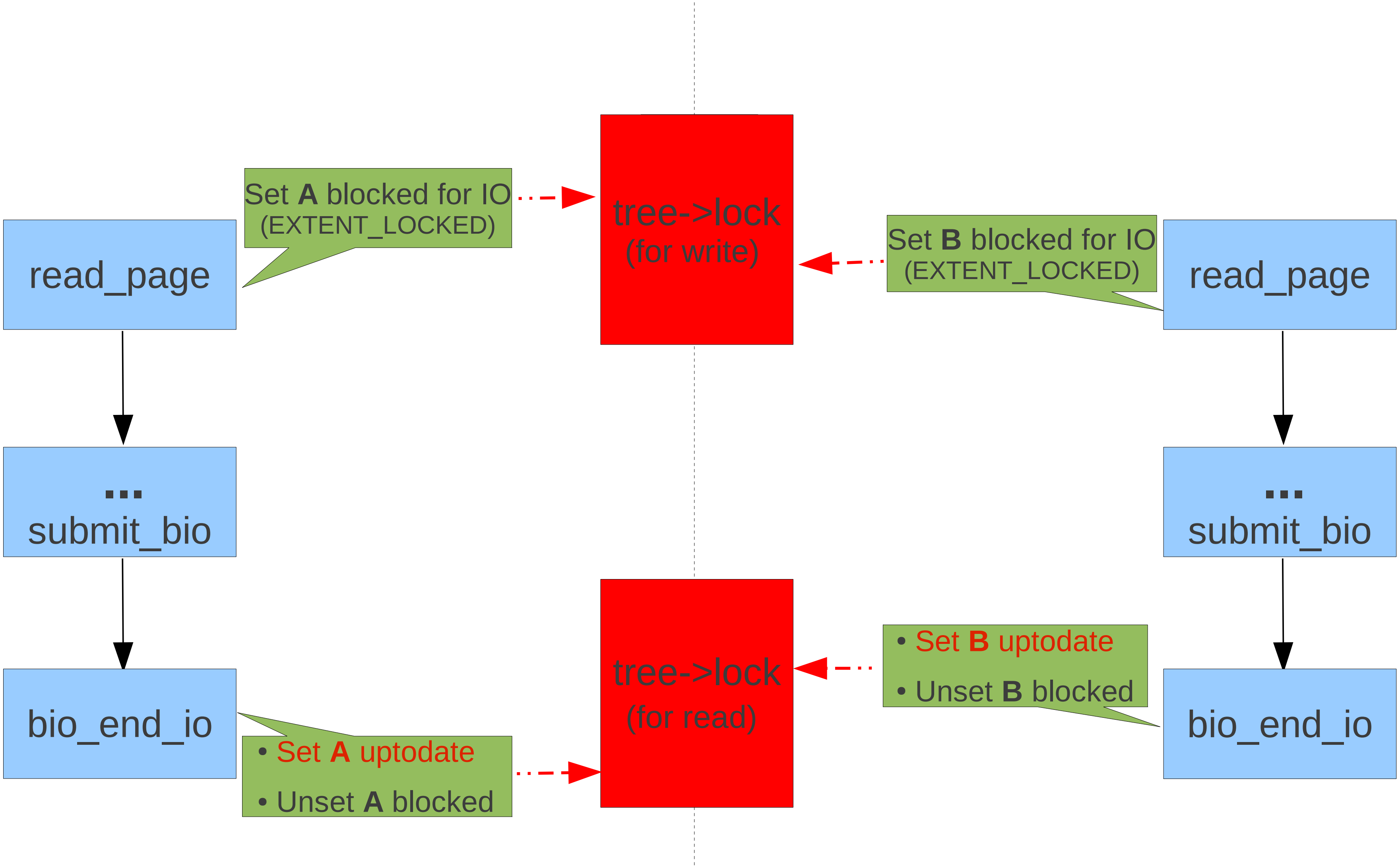


Extent state tree rbroot

A

B

Race on **tree's lock** between **A** and **B**



read_page

Set **A** blocked for IO
(EXTENT_LOCKED)

tree->lock
(for write)

Set **B** blocked for IO
(EXTENT_LOCKED)

read_page

...
submit_bio

...
submit_bio

- Set **B** uptodate
- Unset **B** blocked

bio_end_io

tree->lock
(for read)

bio_end_io

- Set **A** uptodate
- Unset **A** blocked

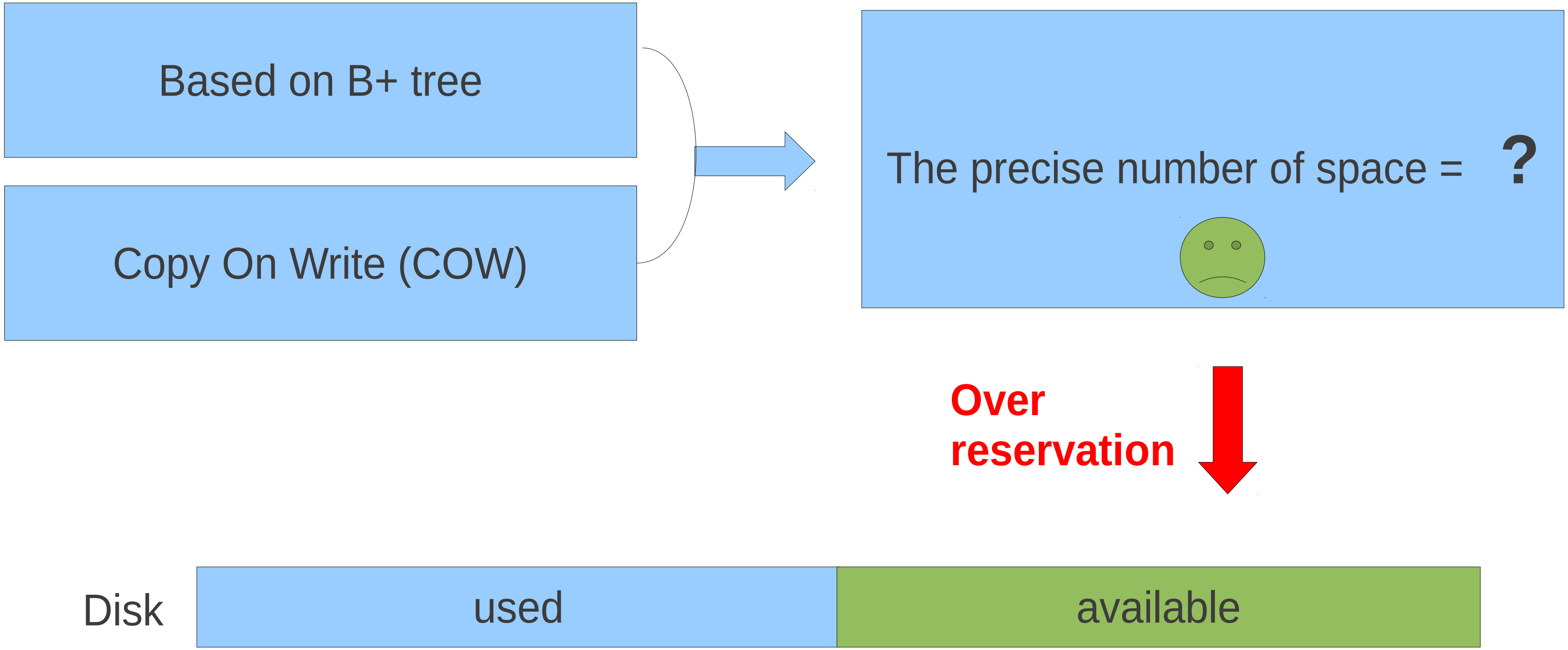
Problems we're facing with

- Over reservation leads to ENOSPC
- Lock contention in kernel data structures
  - Resort to rcu + rbtree / btree / skiplist for lockless read?
  - Reduce lock granularity

Problems we're facing with:
Over reservation leads to ENOSPC

- Btrfs is based on B+ tree
- COW on WAFL
- We are not able to know the precise number of space we're going to use

Because...



Based on B+ tree

Copy On Write (COW)

The precise number of space = ?

Over reservation

Disk

used

available

Then...we have some available space that cannot be allocated :(

Problems we're facing with:

Lock contention in kernel data structures

- Lock contention in in-core rbtrees
  - Extent state tree
  - Free space tree
  - etc
- Possible ways for lockless read
  - Probabilistic skiplist with RCU lock
  - Rbtree with RCU lock
  - Btree with RCU lock
  - Smaller lock granularity

Future work

- Fork a buddy system on space allocation
- Lockless metadata
- Btrfsck (offline/online)
- Performance
  - overall better than ext3 and ext4

The whys and wherefores of using btrfs

- Good performance
- Good scalability
- Good reliability
- High fault tolerance
- Ease of management
- Base stone of distributed file systems like Ceph, etc.

Thanks!

- Thank Liu Bo
- btrfs.wiki.kernel.org