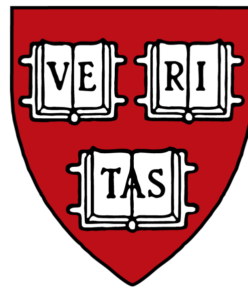
CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu



Lecture 16: Log-structured and journaling filesystems
April 12, 2007

© 2007 Matt Welsh – Harvard University

Log-structured Fileystems (LFS)

Around '91, two trends in disk technology were emerging:

- Disk bandwidth was increasing rapidly (over 40% a year)
- Seek latency not improving much at all
- Machines had increasingly large main memories
  - *Large buffer caches absorb a large fraction of read I/Os*
- Can use for writes as well!
  - *Coalesce several small writes into one larger write*

Some lingering problems with FFS...

- Writing to file metadata (inodes) was required to be synchronous
  - *Couldn't buffer metadata writes in memory*
- Lots of small writes to file metadata means lots of seeks!

LFS takes advantage of both to increase FS performance

- Mendel Rosenblum and John Ousterhout
  - *Mendel is now a prof at Stanford*
- Also lots of contributions by our own Margo Seltzer

© 2007 Matt Welsh – Harvard University

LFS: Basic Idea

Treat the entire disk as *one big append-only log* for writes!

- Don't try to lay out blocks on disk in some predetermined order
- Whenever a file write occurs, append it to the *end* of the log
- Whenever file metadata changes, append it to the *end* of the log

Collect pending writes in memory and stream out in one big write

- Maximizes disk bandwidth
- No "extra" seeks required (only those to move the end of the log)

When do writes to the actual disk happen?

© 2007 Matt Welsh – Harvard University

LFS: Basic Idea

Treat the entire disk as *one big append-only log* for writes!

- Don't try to lay out blocks on disk in some predetermined order
- Whenever a file write occurs, append it to the *end* of the log
- Whenever file metadata changes, append it to the *end* of the log

Collect pending writes in memory and stream out in one big write

- Maximizes disk bandwidth
- No “extra” seeks required (only those to move the end of the log)

When do writes to the actual disk happen?

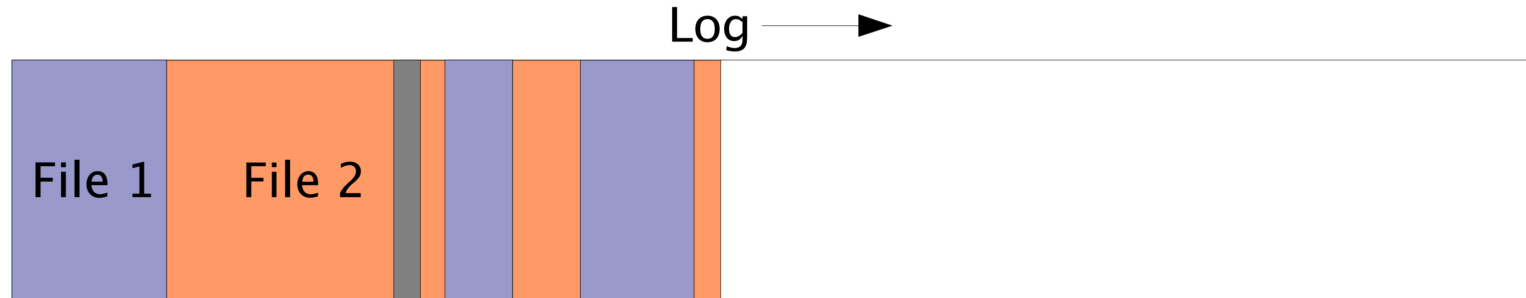
- When a user calls sync() -- synchronize data on disk for whole filesystem
- When a user calls fsync() -- synchronize data on disk for one file
- When OS needs to reclaim dirty buffer cache pages
  - *Note that this can often be avoided, eg., by preferring clean pages*

Sounds simple ...

- But lots of hairy details to deal with!

© 2007 Matt Welsh – Harvard University

LFS Example



Writing a block in the middle of the file just appends that block to the log

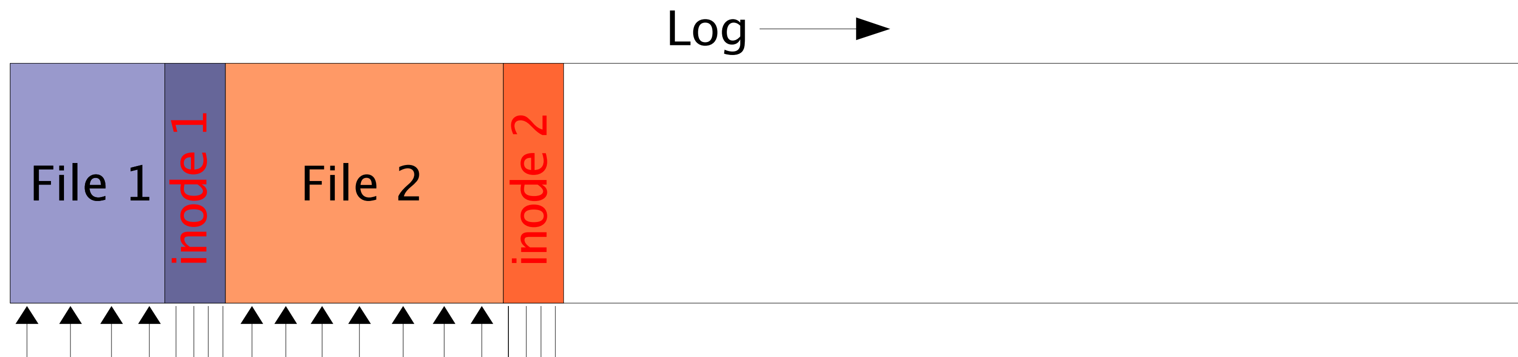
© 2007 Matt Welsh – Harvard University

LFS and inodes

How do you locate file data?

- Sequential scan of the log is probably a bad idea ...

Solution: Use FFS-style inodes!



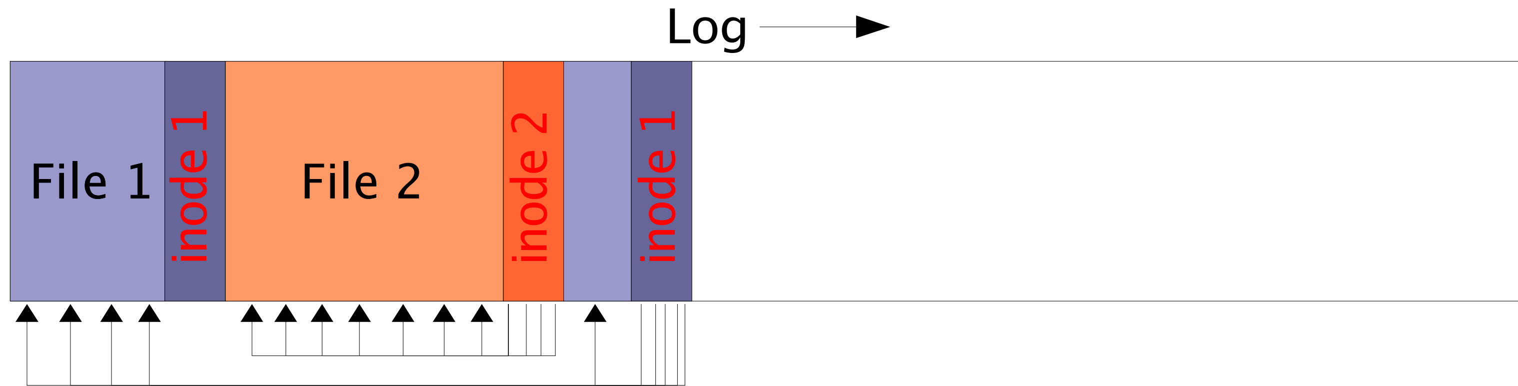
© 2007 Matt Welsh – Harvard University

LFS and inodes

How do you locate file data?

- Sequential scan of the log is probably a bad idea ...

Solution: Use FFS-style inodes!



Every update to a file writes a new copy of the inode!

© 2007 Matt Welsh – Harvard University

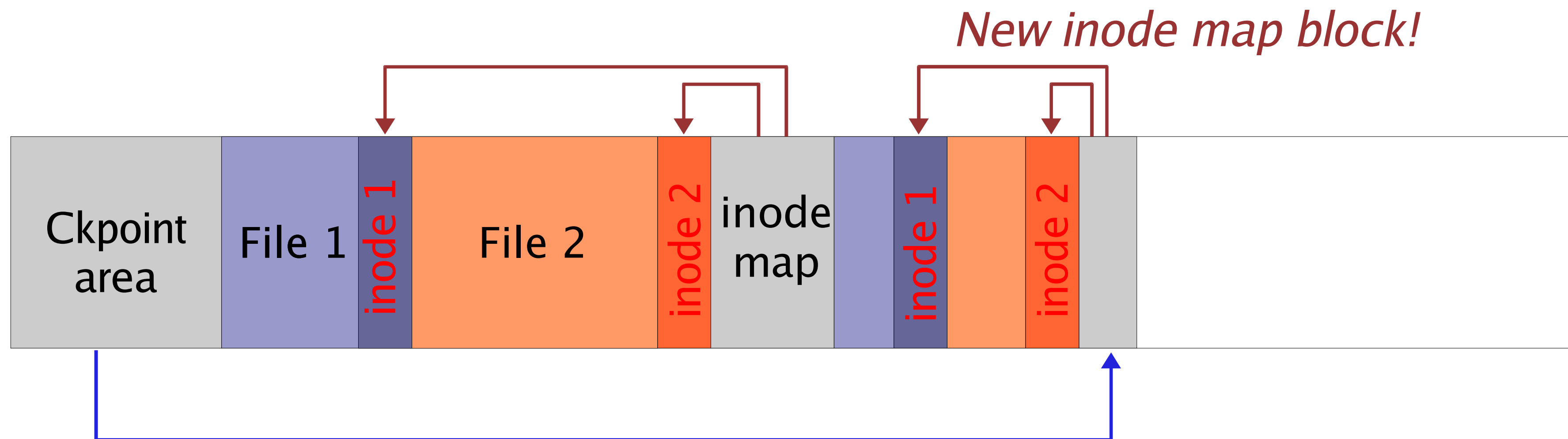
inode map (this is getting fun)

Well, now, how do you find the inodes??

- Could also be anywhere in the log!

Solution: *inode maps*

- Maps "file number" to the location of its inode in the log
- Note that inode map is *also* written to the log!!!!
- Cache inode maps in memory for performance



© 2007 Matt Welsh – Harvard University

Reading from LFS

But wait ... now file data is scattered all over the disk!

- Seems to obviate all of the benefits of grouping data on common cylinders

Basic assumption: Buffer cache will handle most read traffic

- Or at least, reads will happen to data roughly in the order in which it was written
- Take advantage of huge system memories to cache the heck out of the FS!

© 2007 Matt Welsh – Harvard University

Log Cleaner

With LFS, eventually the disk will fill up!

- Need some way to reclaim "dead space"

What constitutes "dead space?"

- Deleted files
- File blocks that have been "overwritten"

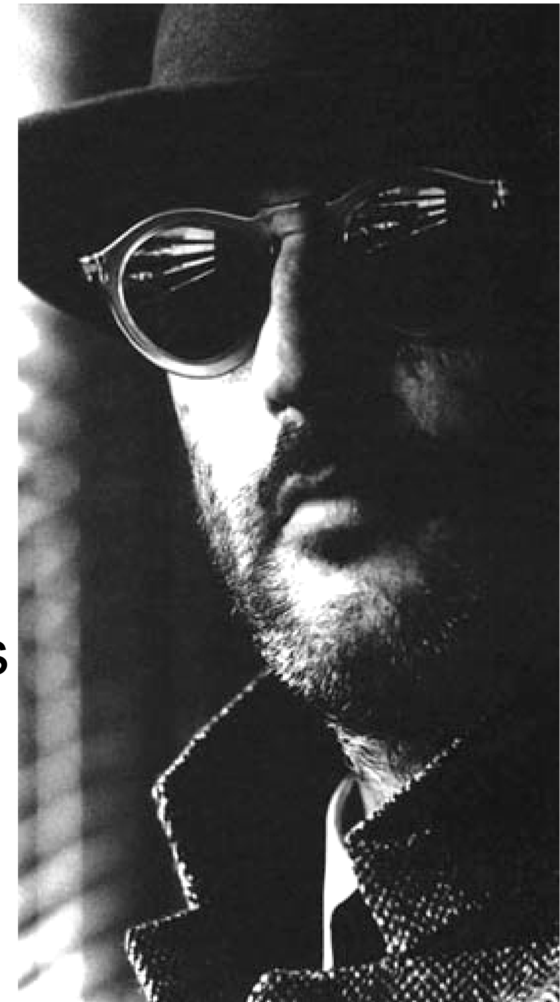
Solution: Periodic "log cleaning"

Scan the log and look for deleted or overwritten blocks

- Effectively, clear out stale log entries

Copy *live data* to the end of the log

- The rest of the log (at the beginning) can now be reused!

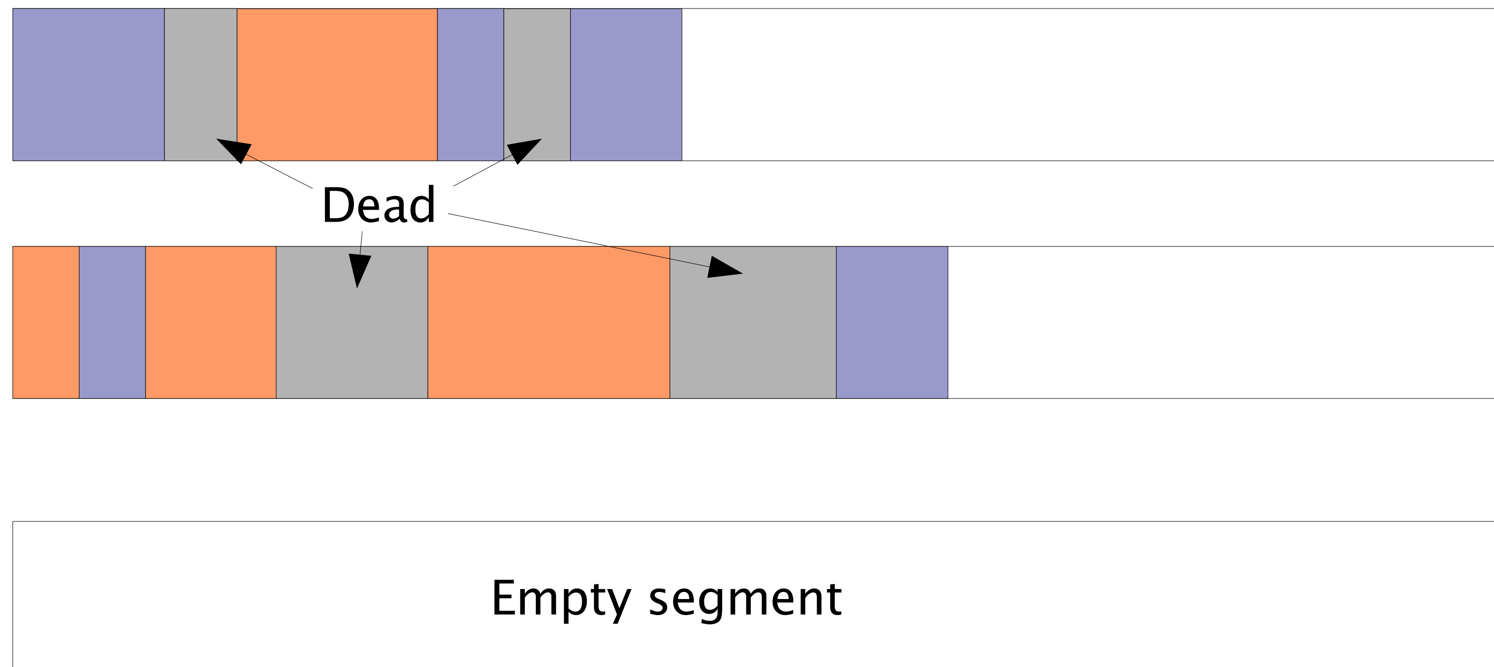


© 2007 Matt Welsh – Harvard University

Log cleaning example

LFS cleaner breaks log into *segments*

- Each segment is scanned by the cleaner
- Live blocks from a segment are copied into a new segment
- The entire scanned segment can then be reclaimed

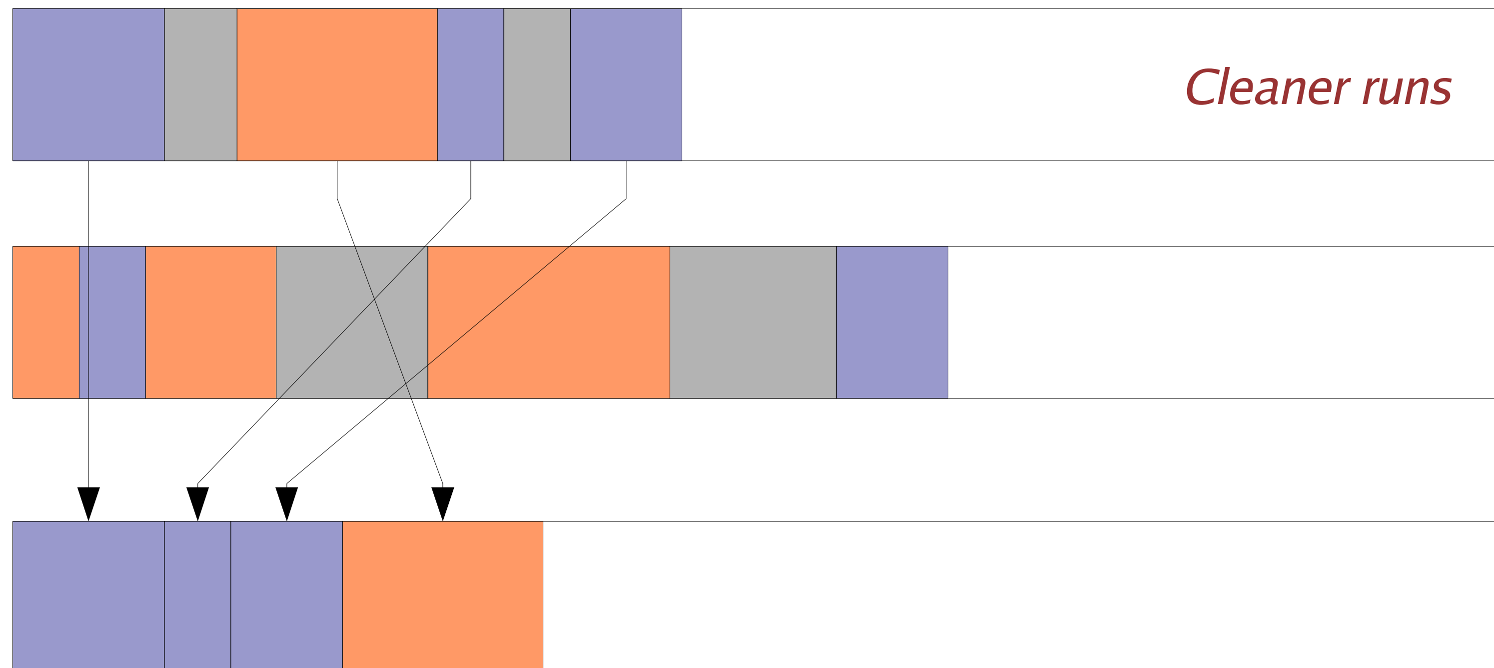


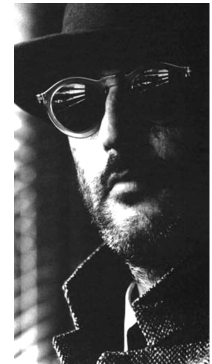
© 2007 Matt Welsh – Harvard University

Log cleaning example

LFS cleaner breaks log into *segments*

- Each segment is scanned by the cleaner
- Live blocks from a segment are copied into a new segment
- The entire scanned segment can then be reclaimed



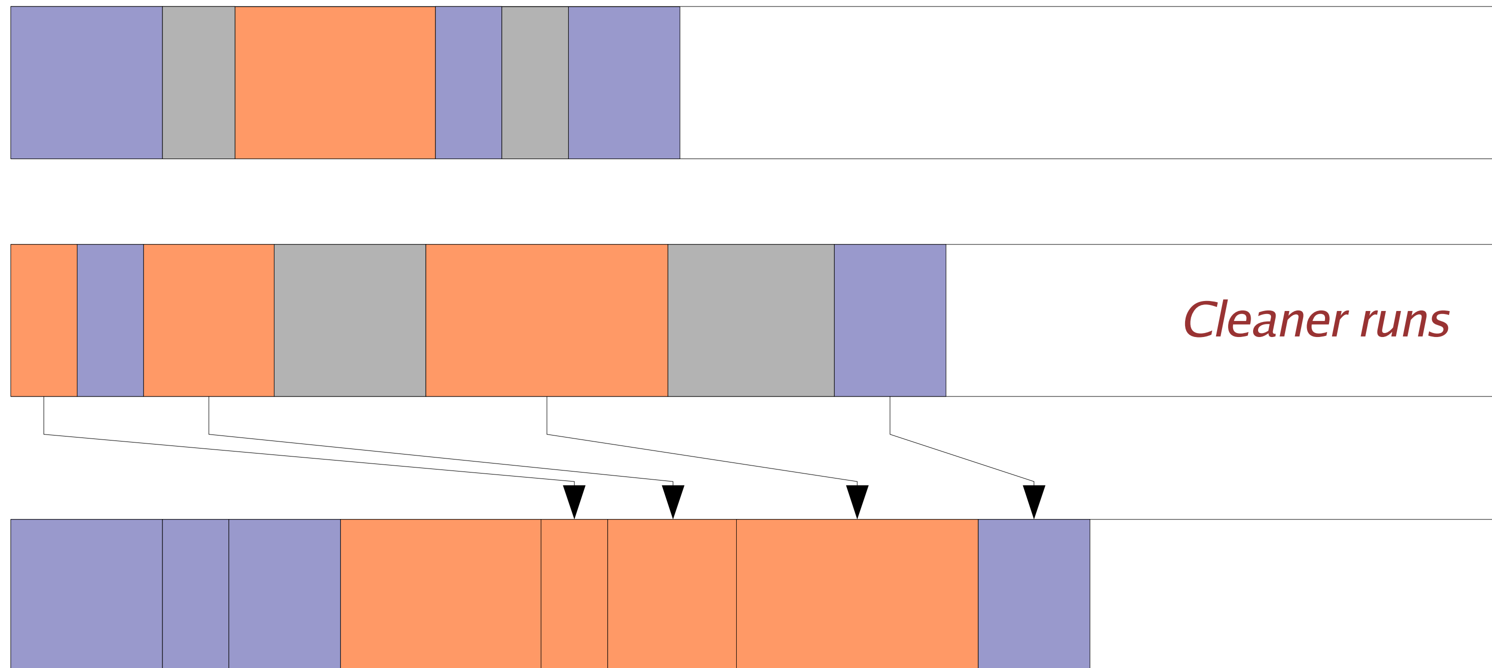


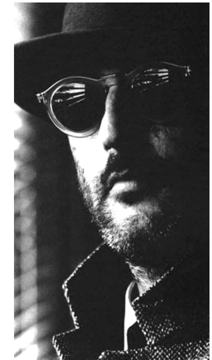
© 2007 Matt Welsh – Harvard University

Log cleaning example

LFS cleaner breaks log into *segments*

- Each segment is scanned by the cleaner
- Live blocks from a segment are copied into a new segment
- The entire scanned segment can then be reclaimed





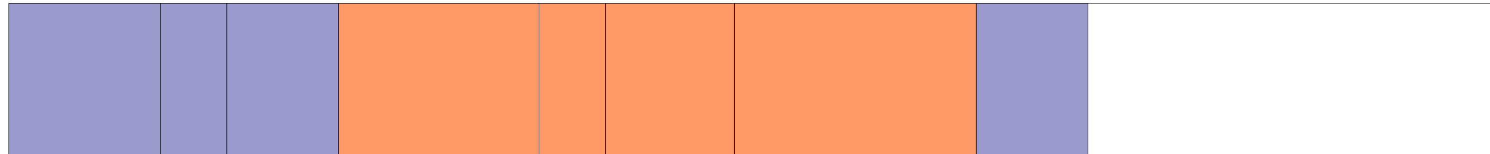
© 2007 Matt Welsh – Harvard University

Log cleaning example

LFS cleaner breaks log into *segments*

- Each segment is scanned by the cleaner
- Live blocks from a segment are copied into a new segment
- The entire scanned segment can then be reclaimed

These two segments are now empty and ready to store new data



© 2007 Matt Welsh – Harvard University

Cleaning Issues

When does the cleaner run?

- Generally when the system (or at least the disk) is otherwise idle
- Can cause problems on a busy system with little idle time

Cleaning a segment requires reading the whole thing!

- Can reduce this cost if the data to be written is already in cache

How does segment size affect performance?

-
-
-

© 2007 Matt Welsh – Harvard University

Cleaning Issues

When does the cleaner run?

- Generally when the system (or at least the disk) is otherwise idle
- Can cause problems on a busy system with little idle time

Cleaning a segment requires reading the whole thing!

- Can reduce this cost if the data to be written is already in cache

How does segment size affect performance?

Large segments amortize cost of access/seek time to read/write entire segment during cleaning

Small segments introduce more variance in segment utilizations

- More segments will contain only dead blocks, making cleaning trivial

Could imagine dynamically chaging segment sizes based on observed overhead for cleaning

© 2007 Matt Welsh – Harvard University

LFS Debate

First LFS paper by Rosenblum and Ousterhout in '91

1992 port of LFS to BSD by Margo Seltzer and others...

Seltzer et al. Publish paper in USENIX'93 pointing out some flaws

Ousterhout publishes critique of '93 LFS paper

Seltzer publishes revised paper in '95

Ousterhout publishes critique of '95 paper

- Seltzer publishes response to critique
  - Ousterhout publishes response to response to critique...

"Lies, damn lies, and benchmarks"

- It is very difficult to come up with definitive benchmarks proving that one system is better than another
- Can always find a scenario where one system design outperforms another
- Difficult to extrapolate based on benchmark tests

© 2007 Matt Welsh – Harvard University

Filesystem corruption

What happens when you are making changes to a filesystem and the system crashes?

- Example: Modifying block 5 of a large directory, adding lots of new file entries
- System crashes while the block is being written
- The new files are "lost!"

System runs `fsck` program on reboot

- Scans through the entire filesystem and locates corrupted inodes and directories
- Can typically find the bad directory, but may not be able to repair it!
  - *The directory could have been left in any state during the write*

`fsck` can take a very long time on large filesystems

- And, no guarantees that it fixes the problems anyway

Journaling Filesystems

Ensure that changes to the filesystem are made *atomically*

- That is, a group of changes are made all together, or not at all

Example: creating a new file

- Need to write both the inode for the new file and the directory entry “together”
- Otherwise, if a crash happens between the two writes, either..
  - 1) Directory points to a file that does not exist
  - 2) Or, file is on disk but not included in any directory

© 2007 Matt Welsh – Harvard University

Journaling Filesystems

Goal: Make updates to filesystems appear to be atomic

- The directory either looks exactly as it did **before** the file was created
- Or the directory looks exactly as it did **after** the file was created
- Cannot leave an FS entity (data block, inode, directory, etc.) in an intermediate state!

Idea: Maintain a **log** of all changes to the filesystem

- Log contains information on any operations performed to the filesystem state
- e.g., "Directory 2841 had inodes 404, 407, and 408 added to it"

To make a filesystem change:

- 1. Write an *intent-to-commit record* to the log
- 2. Write the appropriate **changes** to the log
  - *Do not modify the filesystem data directly!!!*
- 3. Write a *commit record* to the log

This is very similar to the notion of database *transactions*

© 2007 Matt Welsh – Harvard University

Journaling FS Recovery

What happens when the system crashes?

- Filesystem data has not actually been modified, just the log!
- So, the FS itself reflects only what happened *before the crash*

Periodically synchronize the log with the filesystem data

- Called a *checkpoint*
- Ensures that the FS data reflects all of the changes in the log

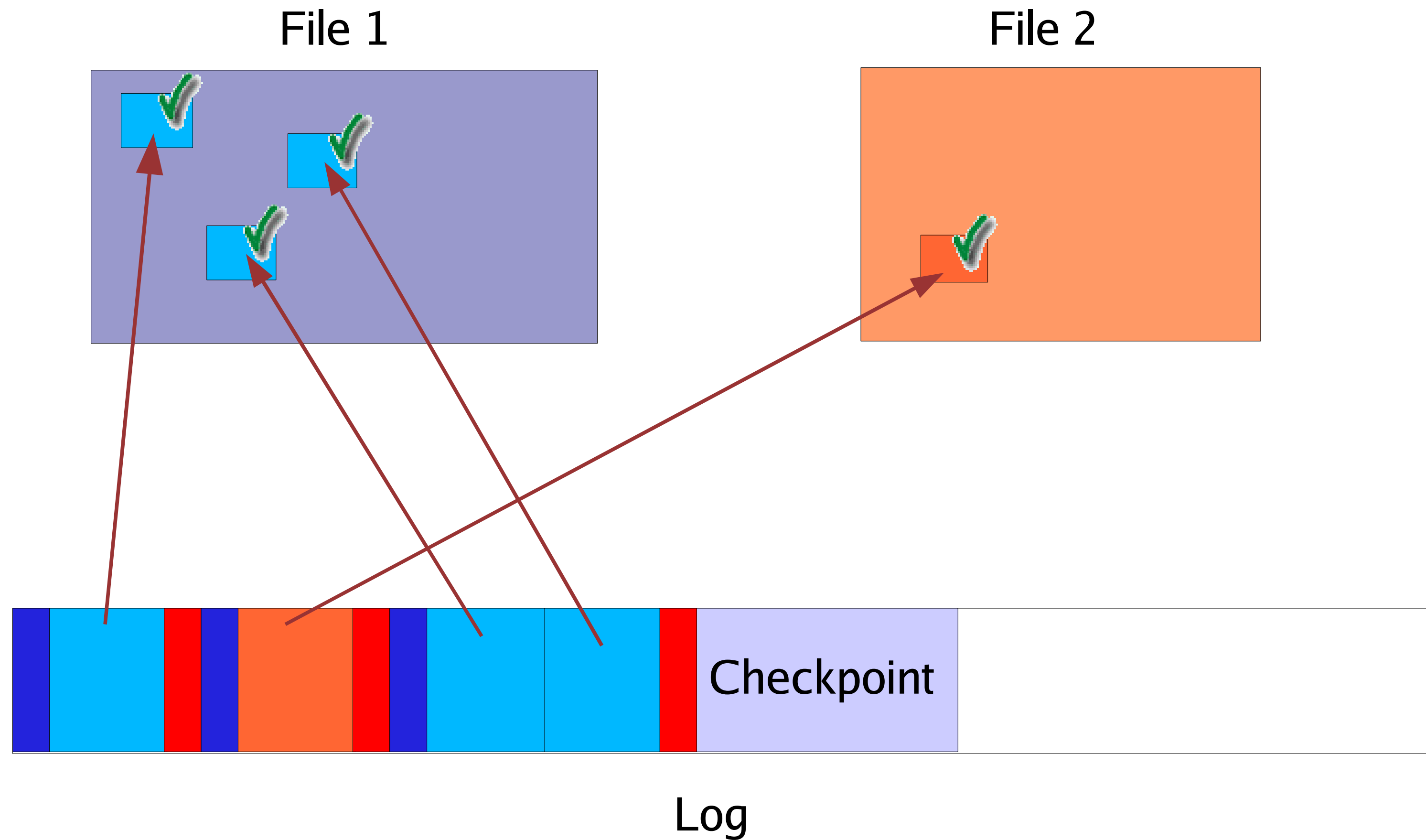
No need to scan the entire filesystem after a crash...

- Only need to look at the log entries **since the last checkpoint**!

For each log entry, see if the commit record is there

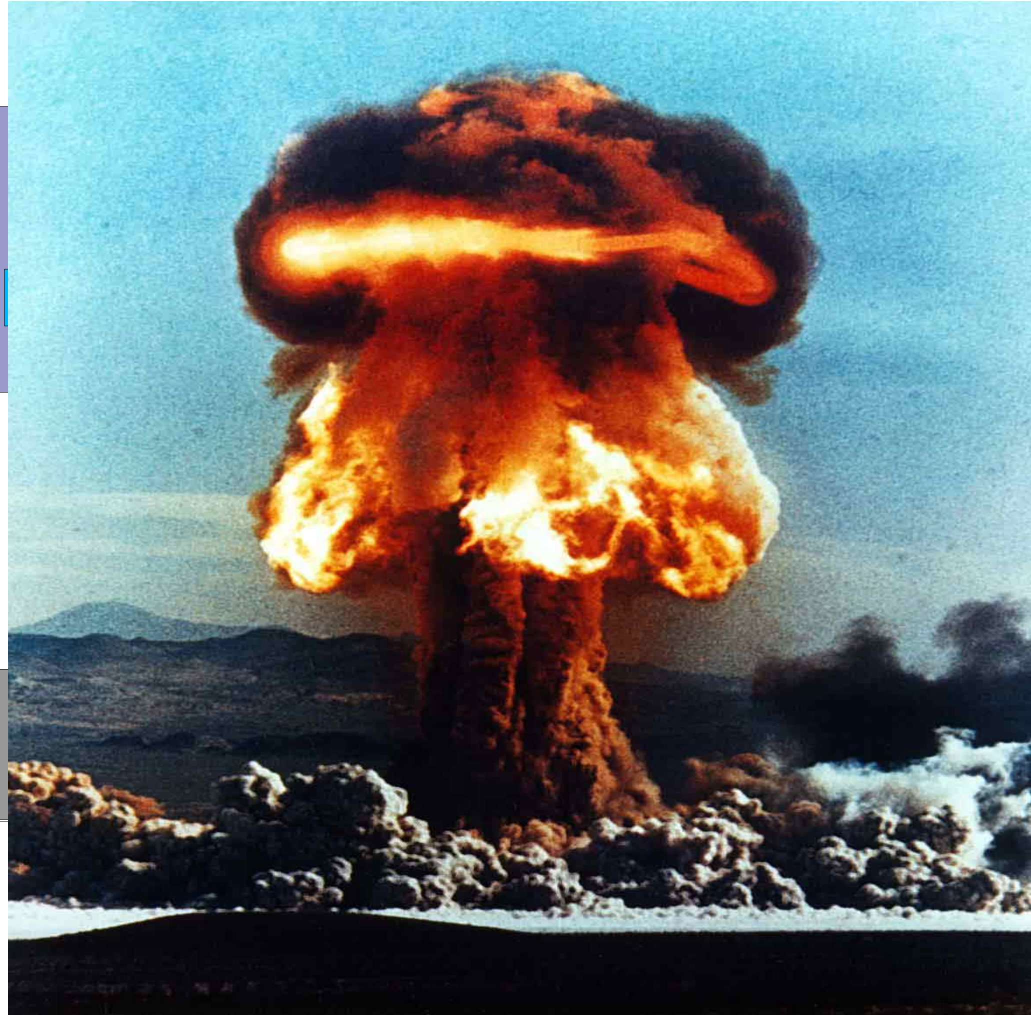
- If not, consider the changes incomplete, and don't try to make them

Journaling FS Example



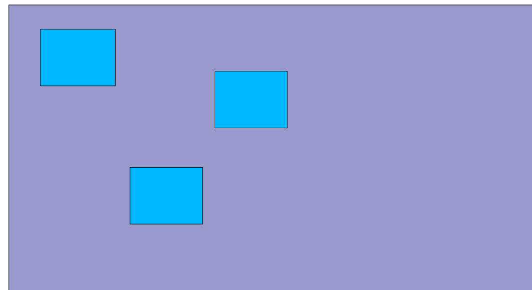
© 2007 Matt Welsh – Harvard University

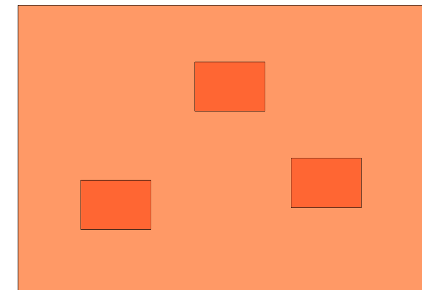
Journaling FS Example

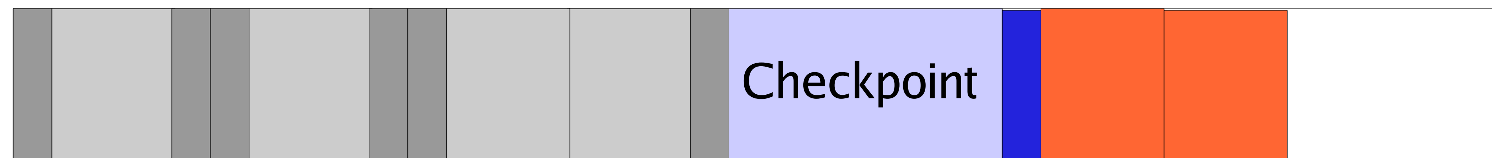


© 2007 Matt Welsh – Harvard University

Journaling FS Example







Filesystem reflects changes up to last checkpoint

Fsck scans changelog from last checkpoint forward
Doesn't find a commit record ... changes are simply ignored

© 2007 Matt Welsh – Harvard University