File Systems

CS 4410
Operating Systems

[R. Agarwal, L. Alvisi, A. Bracy, M. George, E. Sirer, R. Van Renesse]

Logging and LFS

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

The Consistent Update Problem

- Filesystems consist of multiple data structures
  - Free list and directory tree (e.g. inodes)
- Many FS operations require a sequence of updates to multiple data structures
- Need to *atomically* move FS from one valid state to another
- Crashes can occur at any time!

Example: Tiny FFS

- 6 blocks, 6 inodes

Inode bitmap

0	1	0	0	0	0

Data bitmap

0	0	0	0	1	0

i-nodes

	I1				

Data blocks

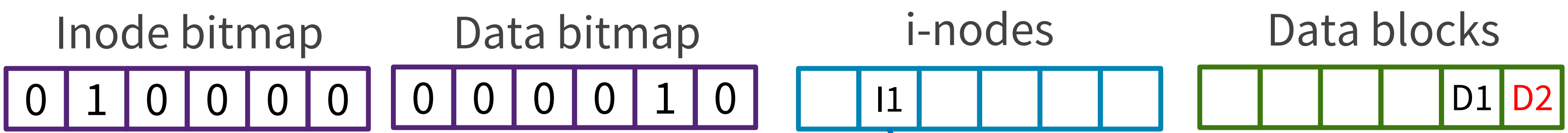
				D1	

We want to append a new data block to the file

Owner:	edward
Permissions:	644
Size:	1
Pointer:	4
Pointer:	null
Pointer:	null
Pointer:	null

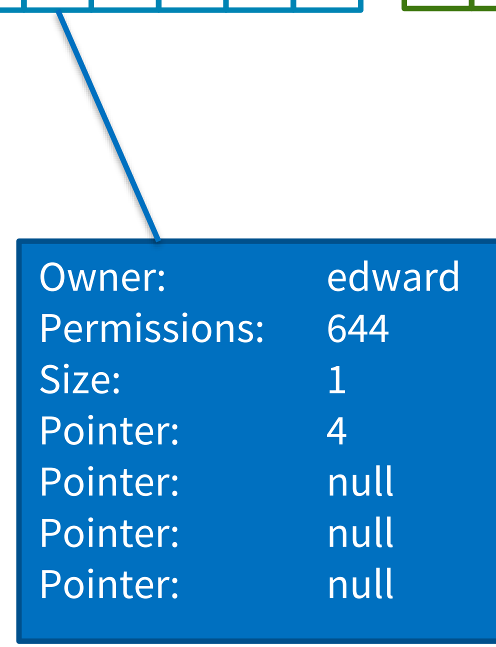
Example: Tiny FFS

- 6 blocks, 6 inodes



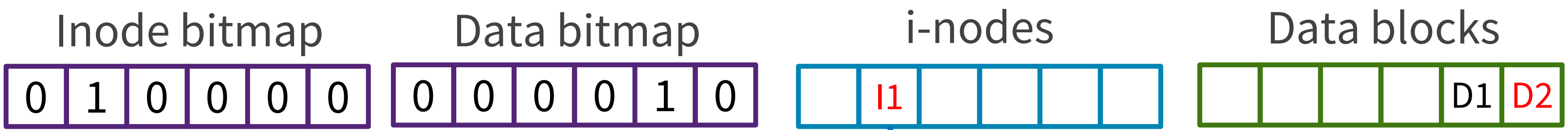
We want to append a new data block to the file

1. Add new data block D2



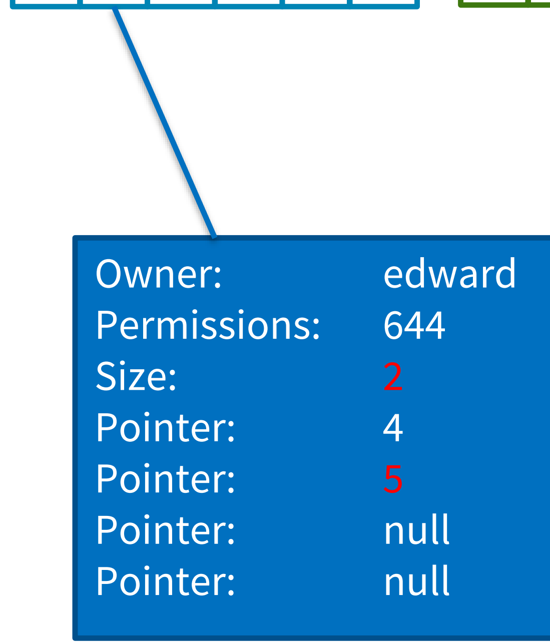
Example: Tiny FFS

- 6 blocks, 6 inodes



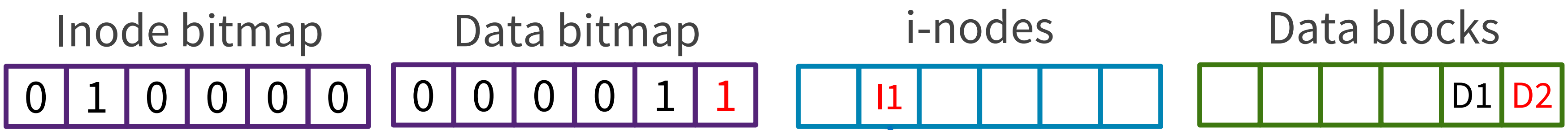
We want to append a new data block to the file

1. Add new data block D2
2. Update inode



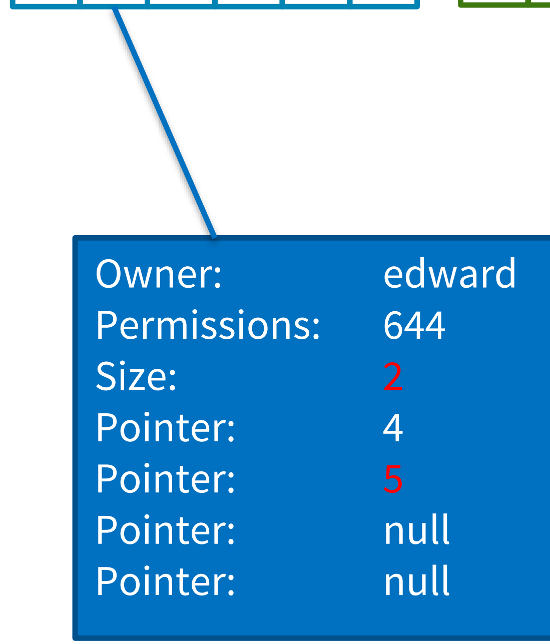
Example: Tiny FFS

- 6 blocks, 6 inodes



We want to append a new data block to the file

1. Add new data block D2
2. Update inode
3. Update free list (data bitmap)



What if a crash/power outage occurs between writes?

Worse: Writes do not occur in order! (C-SCAN?)

When Only One Write Succeeds

- Just the data block (D2) is written to disk
  - Data is written, but no way to get to it – in fact, D2 still appears as a free block
  - Write is lost, but FS data structures are consistent
- Just the updated inode (Iv2) is written to disk
  - If we follow the pointer, we read garbage
  - File system inconsistency: data bitmap says block is free, while inode says it is used
- Just the updated bitmap is written to disk
  - File system inconsistency: data bitmap says data block is used, but no inode points to it. The block will never be used

When Two Writes Succeed

- Inode and data bitmap updates succeed
  - Good news: file system is consistent!
  - Bad news: reading new block returns garbage
- Inode and data block updates succeed
  - File system inconsistency (with free list)
- Data bitmap and data block succeed
  - File system inconsistency
  - No idea which file data block belongs to!

Solution 1: File System Checker

- Upon reboot, scan disk, see if filesystem is in consistent state
- Detect and repair filesystem errors

- Unix: fsck (file system check)
- Windows: scandisk

FSCK Summary

1. Sanity check the superblock
2. Check validity of free block and inode bitmaps
3. Check that inodes are not corrupted
4. Check inode links
5. Check for duplicates
6. Check directories

Checking Validity of Free Bitmaps

- Scan inodes, traversing trees, to see which blocks are allocated
- Build table of blocks, keeping track of status (used or free)
- On inconsistency, overwrite free bitmap

Missing Block 2

(add it to the free list)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
used	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
free list	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1

Checking Inode Links

Use a per-file table instead of per-block

Parse entire directory structure, starting at root

- Increment counter for each file you encounter
- This value can be >1 due to hard links
- Symbolic links are ignored

Compare table counts w/link counts in i-node

- If i-node count > our directory count (wastes space): Reduce i-node count to correct count
- If i-node count < our directory count (catastrophic): Add lost file to lost+found directory

FSCK Summary

1. Sanity check the superblock
2. Check validity of free block and inode bitmaps
3. Check that inodes are not corrupted
4. Check inode links
5. Check for duplicates
6. Check directories

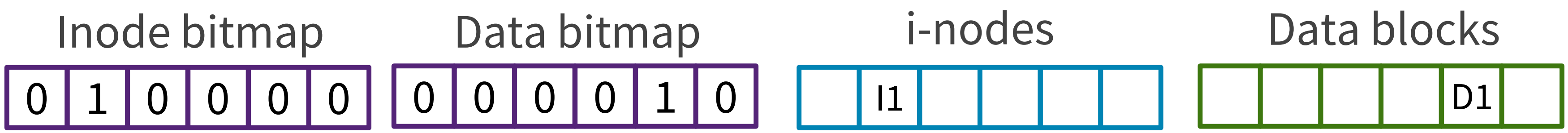
Very Slow, Scales Badly

Solution 2: Journaling

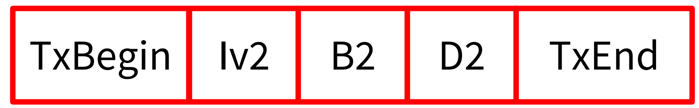
AKA “Write Ahead Logging”

- Turns multiple updates into a single disk write
- Write ahead a short note to a log, specifying changes about to be made to FS data structures
- If a crash occurs while writing to data structures, consult log to determine what to do
  - No need to scan entire disk!

Data Journaling Example



- We want to add a new block to the file
- Three easy steps
  - Write to the log 5 blocks:



    - write each record to a block, so it is atomic
  - Write the blocks for Iv2, B2, D2 to the FS proper
  - Mark the transaction free in the journal
- What if we crash before the log is updated?
  - if no commit, nothing made it into FS - ignore changes!
- What if we crash after the log is updated?
  - replay changes in log back to disk!

Journaling and Write Order

- Issuing the 5 writes to the log sequentially is slow

TxBegin	Iv2	B2	D2	TxEnd

  - Issue at once, and transform into a single sequential write?
- Problem: disk can schedule writes out of order
  - first write TxBegin, Iv2, B2, TxEnd

Crash →

  - then write D2
- Log contains:

TxBegin	Iv2	B2	??	TxEnd

  - syntactically, transaction log looks fine, even with nonsense in place of D2!
- Set a Barrier before TxEnd
  - TxEnd must block until data on disk

Another Approach

Solution 3: Log-Structured FS

- Developed in 1990s
  - Memory got large enough to cache most reads
  - Sequential R/W performance on disks greatly improved, but not random access
  - Lots of seeks to write 1 file kills performance, but this is exactly what FFS requires

- Idea: Use disk as a log
  - Buffer all updates (including metadata!) into an in-memory segment
  - When segment is full, **write to disk** in a long sequential transfer to unused part of disk
- Virtually no seeks
  - much improved disk throughput

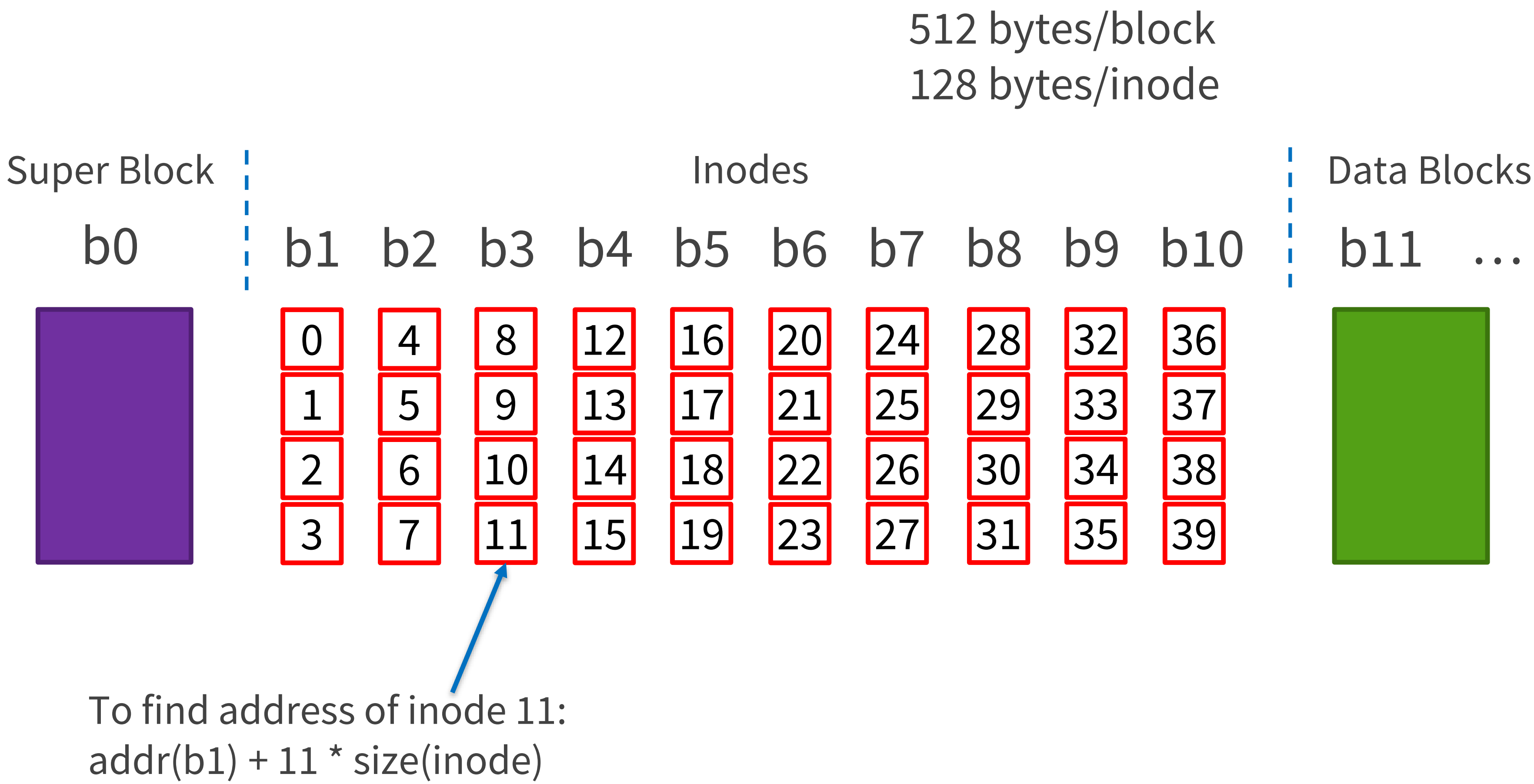
LFS Basics

- Buffered Updates
  - Suppose we want to add a new block to a 0-sized file
  - LFS places both data block and inode in its in-memory segment

- But how do we find the inode?

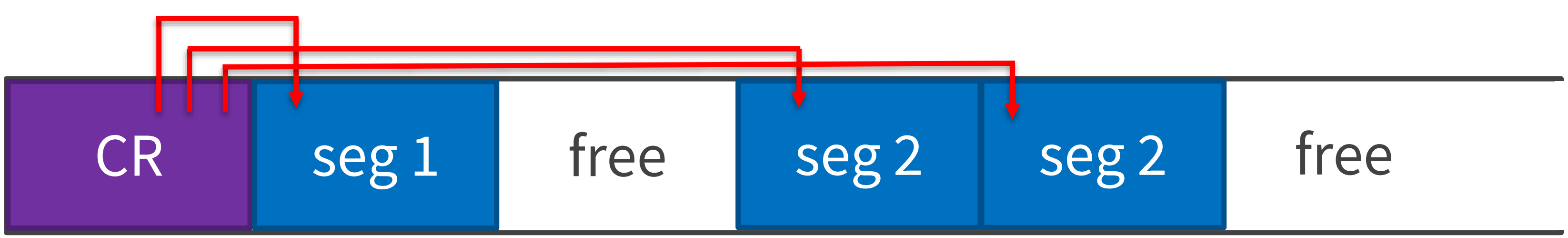
Finding inodes

- In Unix FFS, just index into inode array



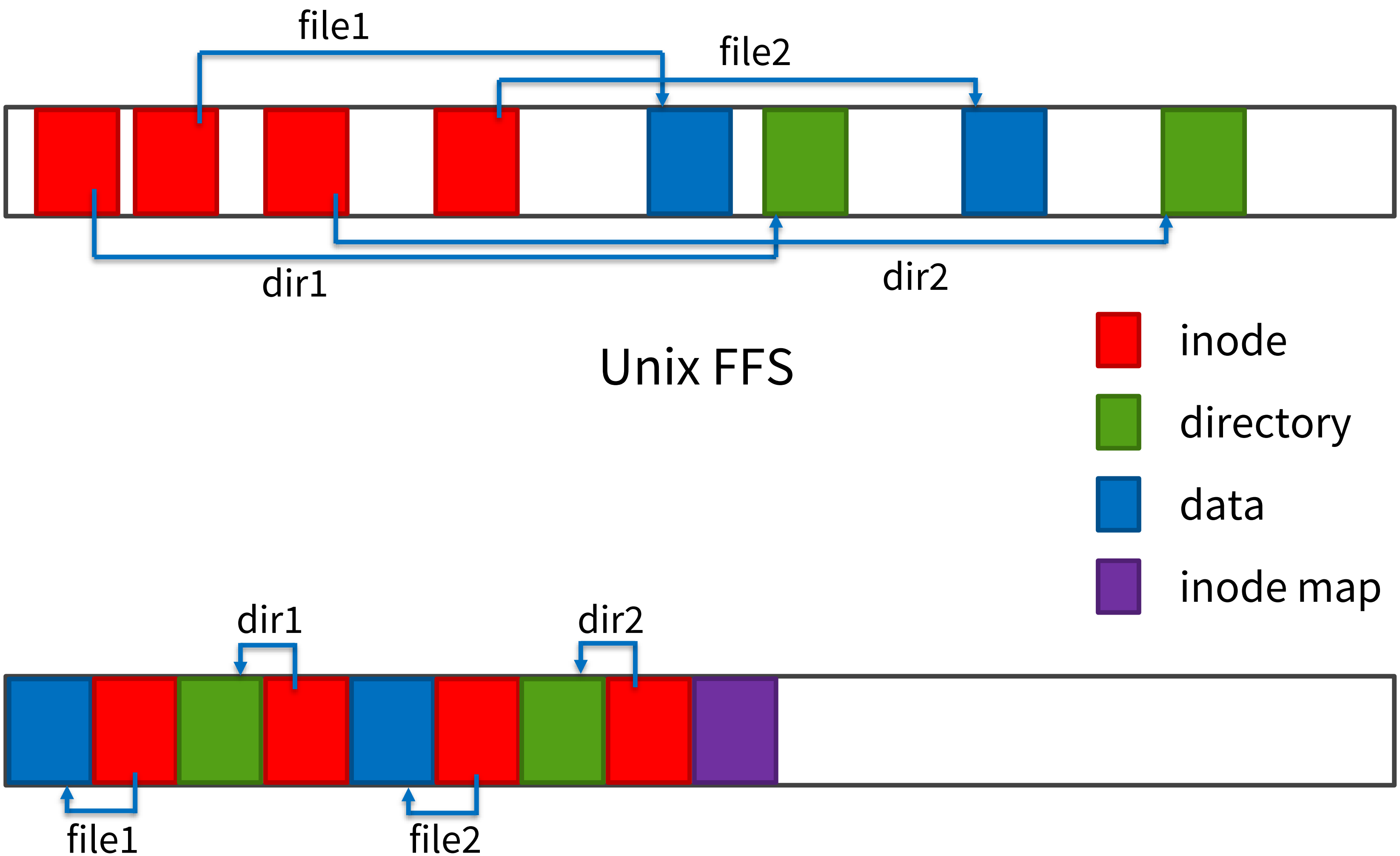
Finding Inodes in LFS

- Inode map: a table indicating where each inode is on disk
  - Normally cached in memory
  - Inode map blocks are written as part of the segment when updated
  - Still no seeking to write to imap ☺
- How do we find the blocks of the Inode map?
  - Listed in a fixed checkpoint region, updated periodically – same function as superblock in FFS



LFS vs FFS

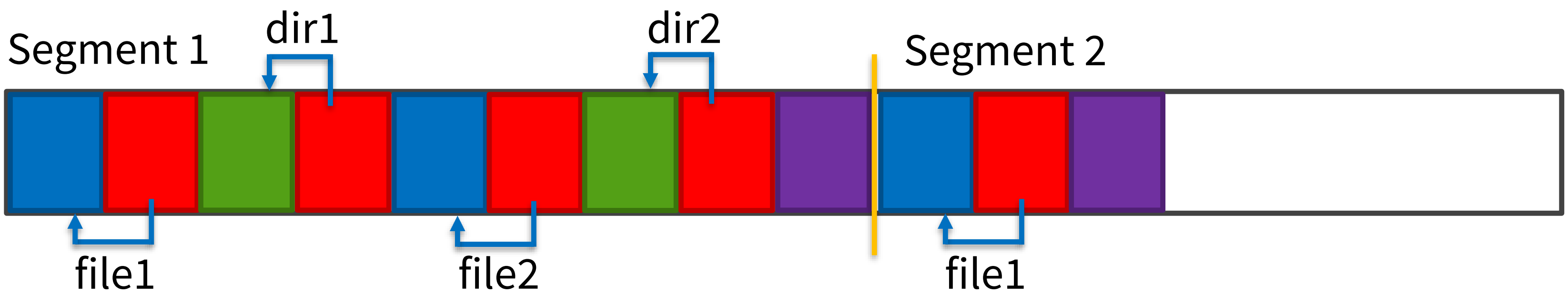
Blocks written to create two 1-block files: dir1/file1 and dir2/file2



Log-Structured FS

Overwriting Data in LFS

- To change data in block 1, create a new block 1
  - Update the inode (create a new one)
  - Update the imap



No need to change dir1, since file1 still has the same inode number

Reading from disk in LFS

- Suppose nothing in memory…
  - read checkpoint region
  - from it, read and cache entire inode map
  - from now on, everything as usual
    - read inode
    - use inode's pointers to get to data blocks
- When the imap is cached, LFS reads involve virtually* the same work as reads in traditional file systems

*modulo an imap lookup

Garbage Collection

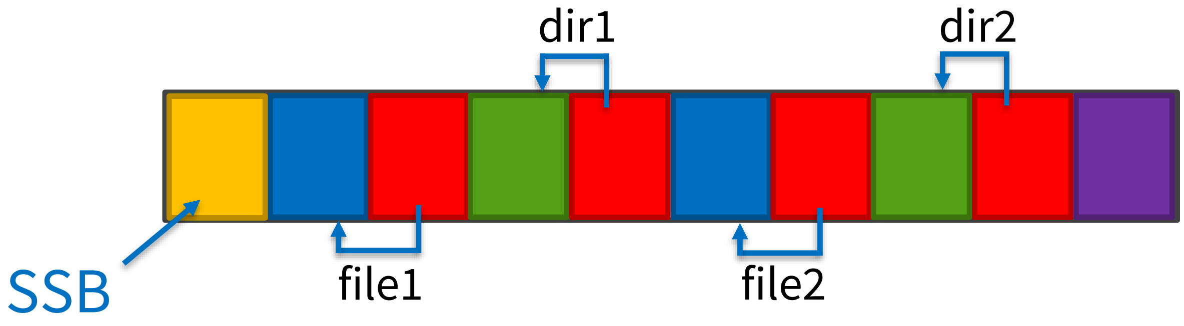
- As old blocks of files are replaced by new, segment in log become fragmented
- Cleaning used to produce contiguous space on which to write
  - compact M fragmented segments into N new segments, newly written to the log
  - free old M segments

Garbage Collection

- Cleaning mechanism:
  - How can LFS tell which segment blocks are live and which dead?
    - Segment Summary Block
    - Replaces free list
- Cleaning policy
  - How often should the cleaner run?
  - How should the cleaner pick segments?

Segment Summary Block

- Kept at the beginning of each segment
- For each data block in segment, SSB holds
  - The file the data block belongs to (inode#)
  - The offset (block#) of the data block within the file



Segment Summary Block

- During cleaning, to determine whether data block D is live:
  - Use inode# to find in imap where inode is currently on disk
  - Read inode (if not already in memory)
  - Check whether pointer for block *block#* refers to D's address
  - If not, D is dead
- Update file's inode with correct pointer if D is live and compacted to new segment

Which segments to clean, and when?

- When?
  - When disk is full
  - Periodically
  - When you have nothing better to do
- Which segments?
  - Utilization: how much is gained by cleaning
    - Segment usage table tracks how much live data in segment; implemented in blocks like inode map
  - Age: how likely is the segment to change soon
    - Better to wait on cleaning a hot segment, since it will quickly get fragmented again

Crash recovery

The journal is the file system!

On recovery:

1. Read checkpoint region
   - May be out of date (written every 30 sec)
   - May be corrupted
     - 1) two CR blocks at opposite ends of disk; 2) timestamp blocks before and after CR
     - use CR with latest consistent timestamp blocks
   - Use latest CR to initialize inode map, segment usage table

Crash recovery

2. Roll forward
   - Start from where checkpoint says log ends (checkpoint region points to last segment)
   - Read through next segments to find valid updates not recorded in checkpoint
     - When a new inode is found, update imap
     - When a data block is found that belongs to no inode, ignore