Operating Systems: Filesystems

Shankar

April 19, 2022

Outline

Overview of filesystem interface

- Persistent structure of files // residing in persistent storage
- Types of files:
  - user file: content is sequence of bytes // u-file for short
  - directory file: content is pointers to files // d-file for short
  - device, ...
- Structure organized as a tree or acylic graph
  - nodes: d-files, u-files
  - root directory: "/"
  - path to a file: "/a/b/. . ."
  - acyclic: more than one path to a u-file (but not directory)
- File metadata: type, owner, creation time, access, ...
- Users can create/delete files, modify content/metadata
- Examples: FAT, UFS, NTFS, ZFS, ..., PFAT, GOSFS, GSFS2

File attributes

- Name
  - last entry in a path to the file // eg, “b” in path “/a/b”
  - subject to size and char limits
- Type: directory or file or device or ...
- Size: subject to limit
- Directory may have a separate limit on number of entries
- Time of creation, modification, last access, ...
- Content type (if u-file): eg, text, binary, executable, ...
  - pdf, jpeg, mpeg, ...
- Owner
- Access for owner, others, ...: eg, r, w, x, setuid, ...

Operations on filesystems

- Format(*dev*)
  - create an empty filesystem on device *dev* (eg, disk, flash, ...)
- Mount(*fstype*, *dev*)
  - attach (to computer) filesystem of type *fstype* on device *dev*
  - returns a path to the filesystem (eg, mount point, volume, ...)
  - after this, processes can operate on the filesystem
- Unmount(*path*)
  - detach (from computer) filesystem at *path* // finish all io
  - after this, the filesystem is inert in its device, unaccessible

Operations on attached filesystem – 1

- `Create(`*path*`)`, `CreateDir(`*path*`)`
  - create a file/directory at given path
- `Link(`*existingPath*`, `*newPath*`)`
  - create a (hard) link to an existing file (not directory)
- `Delete(`*path*`)` // aka `Unlink(`*path*`)`
  - delete the given path to the u-file at *path*
  - delete u-file if no more paths to it
- `DeleteDir(`*path*`)`
  - delete the directory at *path* // must be empty
- Change attributes (name, metadata) of file at *path*
  - eg, `stat`, `touch`, chown/chgrp, chmod, `rename`/mv

Operations on attached filesystem – 2

- `Open(`*path*, *access*`)`, `OpenDir(`*path*, *access*`)`
  - open the file at *path* with given *access* (r, w, ...)
  - returns a file descriptor
  - after this, file can be operated on
- `Close(`*fd*`)`, `CloseDir(`*fd*`)`
  - close the file associated with file descriptor *fd*
- `Read(`*fd*, *file range*, *buffer*`)`, `ReadDir(`*fd*, *dir range*, *buffer*`)`,
  - read the given range from open file *fd* into given buffer
  - returns number of bytes/entries read
- `Write(`*fd*, *file range*, *buffer*`)`
  - write buffer contents into the given range of open file *fd*
  - returns number of bytes written

Operations on attached filesystem – 3

- Seek(*fd*, *file location*), SeekDir(*fd*, *entry*)
  - move "r/w" position" to given location/entry
- MemMap(*fd*, *file range*, *mem range*)
  - map the given range of open file *fd* to given range of memory
- MemUnmap(*fd*, *file range*, *mem range*)
  - unmap the given range of file *fd* from given range of memory
- Sync(*fd*)
  - complete all pending io for open file *fd*

Consistency of shared files

- Shared file: file opened by several processes concurrently
- Consistency:
  - when does a read see the result of a previous write by another process
- Various types of consistency (from strong to weak)
  - when the read starts after the write returns
  - when the read starts after a post-write sync returns
  - when the read starts after a post-write close returns
- Single-processor system
  - all types of consistency easily achieved
- Multi-processor system
  - strong notions are expensive/slow to achieve

Reliability

- Filesystem should be resilient to device failures
- Types of failures to be handled:
  - failures in persistent storage devices:
    - magnetic / mechanical / electronic parts wear out
  - Operating system may crash in the middle of a fs operation
  - Power loss in the middle of a fs operation
- "Small" failures should cause no loss of filesystem
- "Large" failures may cause loss of some files but no inconsistency (no undetected corrupted files)

Outline

Disks: Geometry

- Platter(s) fixed to rotating spindle
- Spindle speed
- Platter has two surfaces
- Surface has concentric tracks
- Track has sectors
  - more in outer than inner
- Sector: fixed capacity
- Movable arm with fixed rw heads, one per surface
- Buffer memory

Eg, laptop disk (2011)

- 1 or 2 platters
- 4200–15000 rpm, 15–4 ms/rotation
- diameter: 2.5 in
- track width: $<$ 1 micron
- sector: 512 bytes
- buffer: 16 MB

Disk IO

- IO is in blocks (sectors); slower, more bursty than memory
- Disk access time = seek + rotation + transfer
- Seek time: (moving rw head) + (electronic settling)
  - minimum: target is next track; settling only
  - maximum: target is at other end
- rotation delay: half-rotational delay on avg
- transfer time: platter ↔ buffer
- transfer time: buffer ↔ host memory

Eg, laptop disk (2011)

- min seek: 0.3–1.5 ms
- max seek: 10–20 ms
- rotation delay: 7.5–2 ms
- platter ↔ buffer: 50 (inner) – 100 (outer) MB/s
- buffer ↔ host memory: 100–300 MB/s

Disk: Scheduling IO requests

- FIFO: terrible: lots of head movement
- SSTF (shortest seek time first)
  - favors "middle" requests; can starve "edge" requests
- SCAN (elevator)
  - sweep from inner to outer, until no requests in this direction
  - sweep from outer to inner, until no requests in this direction
- CSCAN: like SCAN but in only one direction
  - fairer, less chance of sparsely-requested track
- R-SCAN / R-CSCAN
  - allow minor deviations in direction to exploit rotational delays

Outline

Overview of Fs Implementation – 1

- Define a mapping of filesystem files to device blocks
- Define an implementation of filesystem operations
  - performance: random and sequential data access
  - reliability: inspite of device failures
  - accomodate different devices (blocks, size, speed, geometry, ...)

- The slides sometimes use the following abbreviations:
  - fs: filesystem
  - fs-int: filesystem interface
  - fs-imp: filesystem implementation

Overview of Fs Implementation – 2

- Fs implementation usually starts at the second block in the device

  [The first device block is usually reserved for the boot sector]

- Fs implementation organizes the device in fs blocks 0, 1, $\cdots$
- One fs block is one or more device blocks, or vice versa
  - henceforth, "block" without qualifier means "fs block"

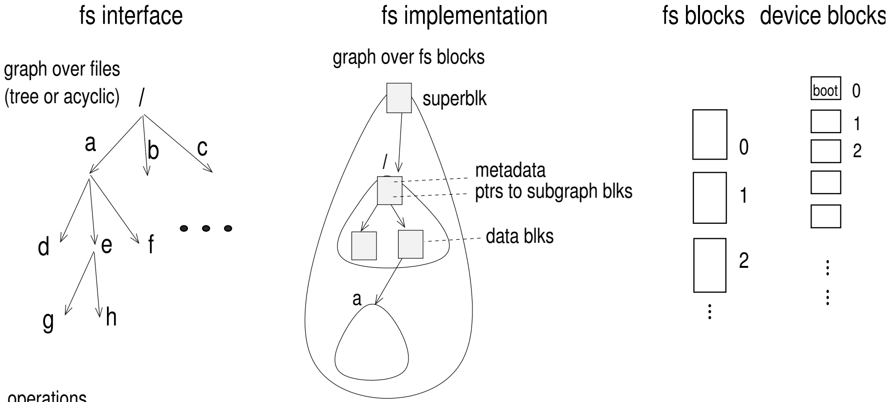
- Fs implementation is a graph over blocks, rooted at a special block (the superblock)

  [In contrast, fs interface is a graph over files]

- Each fs-int file x maps to a subgraph of the fs-imp graph
  - subgraph's blocks hold x's metadata and x's data
  - root block of the subgraph typically has pointers to subgraph's blocks

- List of free blocks reachable from superblock

Overview of Fs Implementation – 3



operations
- mount, unmout, ...
- create, link, unlink, delete
- open, read, write, sync, close

implementations of operations
garbage collection

Superblock

- Usually in the first few device blocks after the boot sector
- First fs block read by OS when mounting a filesystem
- Contains info sufficient to mount the filesystem
  - magic number, indicating filesystem type
  - fs blocksize (vs device blocksize)
  - size of disk (in fs blocks).
  - pointer (block #) to the root of “/” directory’s subgraph
  - pointer to list of free blocks
  - (perhaps) pointer to array of roots of subgraphs
  - ...

Subgraph for a fs-int file

- Unique low-level name // typically a number ("inode number")
- User name(s) // multiple names/paths due to links
- File metadata
  - size
  - owner, access, ...
  - creation time, last modification time, ...
  - ...
- Pointers to fs blocks containing file's data
  - pointers organized in array, linked-list, ...
- For a u-file, the data is the file's data
- For a d-file, the data is a table of directory entries
  - table may be unordered, sorted, hashed, ..., depending on number and size of entries, desired performance, ...
  - directory entry points to the entry's subgraph // eg, inode

To achieve good performance

- Want large numbers of consecutive reads or writes
- So put related info in nearby blocks / cylinders
- Large buffers (to minimize read misses)
- Batched writes: large queue of writes

Reliability: Overcoming device block failures

- A device block can degrade over time.
  - positions in the block may not retain their values
  - need to detect the degradation and avoid that block
- Redundancy within a disk
  - error-detection/correction code (EDC/ECC) in each disk block
  - map each fs block to multiple disk blocks
  - dynamically remap within a disk to bypass failing areas
- Redundancy across disks // eg, RAID
  - map each fs block to blocks in different disks
  - EDC/ECC for a fs block across disks
  - ...

Atomicity of fs operations

- A fs operation is atomically executed if either all of it or none of it is applied to the fs
- Goal: every fs operation is atomic inspite of failures (OS, power, etc) during operation
- Assumption about operations (reads, writes) on device blocks
  - if there is a failure during a block write, the write is completed or the block is unchanged
  - if a sequence of operations is submitted, when the disk indicates completion, all the operations have been done in some order

Atomicity via Copy-On-Write

- Suppose user modifies a file f
- Identify a subgraph, say X, of the fs-imp graph to be modified
- Write the new value of X in fresh blocks, say Y
- Attach Y to the fs-imp graph in place of X
  - typically involves modifying fewer blocks, so low prob of failure
  - ideal: involves modifying one device block
- Garbage collect the blocks of X

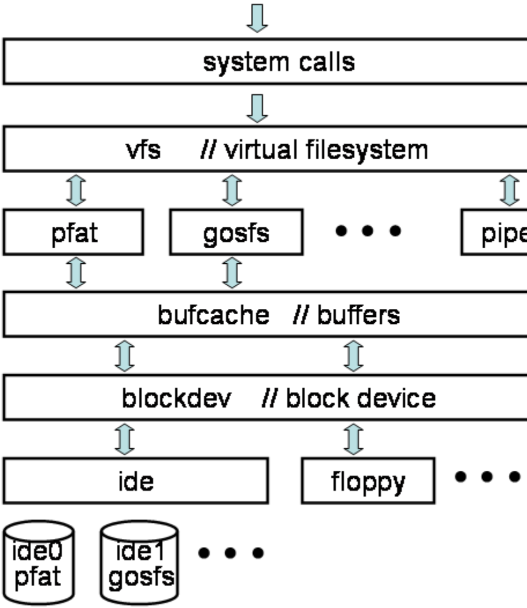
Atomicity via Logging / Journal

- Suppose user issues a sequence of disk operations
- Maintain a log (aka journal) of requested operations
  - add records (one for each operation) to log
  - add "commit" record after last operation
- Later, commit the log to disk
  - write the operations in the log to disk
  - when those writes are completed, erase the log
- Upon recovery from crash, (re)do all operations in the log
  - writes may be repeated, but this is ok // writes are idempotent

Hierarchy in Filesystem

- Virtual filesystem: optional
  - memory-only framework on which to mount real filesystems
- Mounted Filesystem(s)
  - real filesystems, perhaps of different types
- Block cache
  - cache filesystem blocks: performance, sharing, ...
- Block device
  - wrapper for the various block devices with filesystems
- Device drivers for the various block devices

GeekOS: Hierarchy in Filesystem



Outline

FAT32 Overview – 1

- FAT: MS-DOS filesystem
  - simple, but no good for scalability, hard links, reliability, ...
  - currently used only on simple storage devices: flash, ...

- Disk divided into following regions
  - Boot sector: device block 0
    - BIOS parameter block
    - OS boot loader code
  - Filesystem info sector: device block 1
    - signatures, fs type, pointers to other sections
    - fs blocksize, # free fs blocks, # last allocated fs block
    - ...
  - FAT: fs blocks 0 and 1; corresponds to the superblock
  - Data region: rest of the disk, organized as an array of fs blocks
    - holds the data of the fs-int files

FAT32 Overview – 2

- Each block in the data region is either free or bad or holds data (of a file or directory)

- FAT: array with an entry for each block in data region
  - entries $j_0, j_1, \cdots$ form a chain iff blocks $j_0, j_1, \cdots$ hold successive data of a file

- Entry n contains
  - constant, say FREE, if block n is free
  - constant, say BAD, if block n is bad (ie, unusable)
  - 32-bit number, say x, if block n holds data of a file and block x holds the succeeding data of the file
  - constant, say END, if block n holds the last data chunk

- Root directory table: typically at start of data region (block 2)

FAT32 Overview – 3

- Directory entry: 32 bytes
  - name (8)
  - extension (3)
  - attributes (1)
    - read-only, hidden, system, volume label, subdirectory, archive, device
  - reserved (10)
  - last modification time (2)
  - last modification date (2)
  - fs block # of starting fs block of the entry's data
  - size of entry's data (4)
- Hard links??

Outline

1. Filesystem Interface
2. Persistent Storage Devices: Hard disks
3. Filesystem Implementation
4. FAT Filesystem
5. FFS: Unix Fast Filesystem
6. NTFS: Microsoft Filesystem
7. Crash Consistency (OSTEP 42)
8. Log-structured Filesystems (OSTEP 43)
9. Persistent Storage Devices: Flash-based SSDs (OSTEP 44)

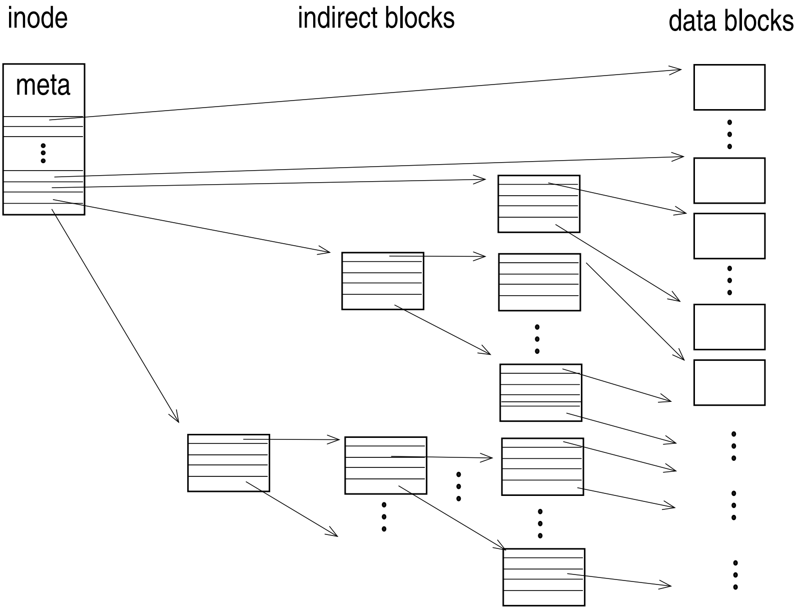
FFS layout

- Boot blocks // few blocks at start
- Superblock // after boot blocks
  - magic number
  - filesystem geometry (eg, locations of groups)
  - filesystem statistics/tuning params
- Groups, each consisting of // cylinder groups
  - backup copy of superblock
  - header with statistics
  - array of inodes
    - holds metadata and data pointers of fs-int files
    - # inodes fixed at format time
  - array of data blocks
  - free-inodes bitmap, free-datablocks bitmap
  - ...

FFS inodes

- Inodes are numbered sequentially starting at 0
  - inodes 0 and 1 are reserved
  - inode 2 is the root directory's inode
- An inode is either free or not
- Non-free inode holds metadata and data pointers of a file
  - owner id
  - type (directory, file, device, ...)
  - access modes
  - reference count // # of hard links
  - size and # blocks of data
  - 15 pointers to data blocks
    - 12 direct
    - 1 single-indirect, 1 double-indirect, 1 triple-indirect

FFS data pointers: asymmetric tree (max depth 4)



FFS directory entries

- The data blocks of a directory hold directory entries
- A directory entry is not split across data blocks
- Directory entry has a pointer to inode
- Directory entry for a file
  - # of the inode of the file // hard link
  - size of entry
  - length of file name (up to 255 bytes)
  - entry name
- Multiple directory entries can point to the same inode

User Ids

- Every user account has a user id (uid)
- Root user (aka superuser, admin) has uid of 0
- Processes and filesystem entries have associated uids
  - indicates owners
  - determines access processes have to filesystem entries
  - determines which processes can be signalled by a process

Process Ids

- Every process has two associated uids
  - effective user id (euid)
    - uid of user on whose behalf it is currently executing
    - determines its access to filesystem entries
  - real uid (ruid)
    - uid of the process's owner
    - determines which processes it can signal:
      x can signal y only if x is superuser or x.ruid $=$ y.ruid

Process Ids

- Process is created: ruid/euid $\leftarrow$ creating process's euid
- Process with euid 0 executes `SetUid(z)`: ruid/euid $\leftarrow$ z
  - no effect if process has non-zero euid
- Example `SetUid` usage
  - login process has euid 0 (to access auth info files)
  - upon successful login, it starts a shell process (with euid 0)
  - shell executes `SetUid(`*authenticated user's uid*`)`
- When a process executes a file `f` with "setuid bit" set:
  its euid is set to `f`'s owner's uid while it is executing `f`.
- Upon bootup, the first process ("init") runs with uid of 0
  - it spawns all other processes directly or indirectly

Directory entry's uids and permissions

- Every directory entry has three classes of users:
  - owner (aka "user")
  - group (owner need not be in this group)
  - others (users other than owner or group)
- Each class's access is defined by three bits: r, w, x
- For a file:
  - r: read the file
  - w: modify the file
  - x: execute the file
- For a directory:
  - r: read the names (but not attributes) of entries in the directory
  - w: modify entries in the directory (create, delete, rename)
  - x: access an entry's contents and metainfo
- When a directory entry is created: attributes are set according to the creating process's attributes (euid, umask, etc)

Directory entry's setuid bit

- Each directory entry also has a "setuid" bit.
- If an *executable file* has setuid set and a process (with execute access) executes it, the process's euid changes to the file's owner's uid while executing the file.
- Typically, the executable file's owner is root, allowing a normal user to get root privileges while executing the file
- This is a high-level analog of system calls

Directory entry's sticky bit

- Each directory entry also has a **sticky** bit.
- Executable file with sticky bit set: hint to the OS to retain the text segment in swap space after the process executes
- An entry x in a directory with sticky bit set:
  - a user with wx access to the directory can rename/delete an entry x in the directory only if it is x's owner (or superuser)
  - Usually set on /tmp directory.

Directory entry's setgid bit

- Unix has the notion of groups of users
- A group is identified by a group id, abbreviated gid
- A gid defines a set of uids
- A user account can be in different groups, i.e., have multiple gids

- Process has effective gid (egid) and real gid (rgid)
  - play a similar role as euid and ruid

- A directory entry has a setgid bit
  - plays a similar role to setuid for executables
  - plays an entirely different role for directories

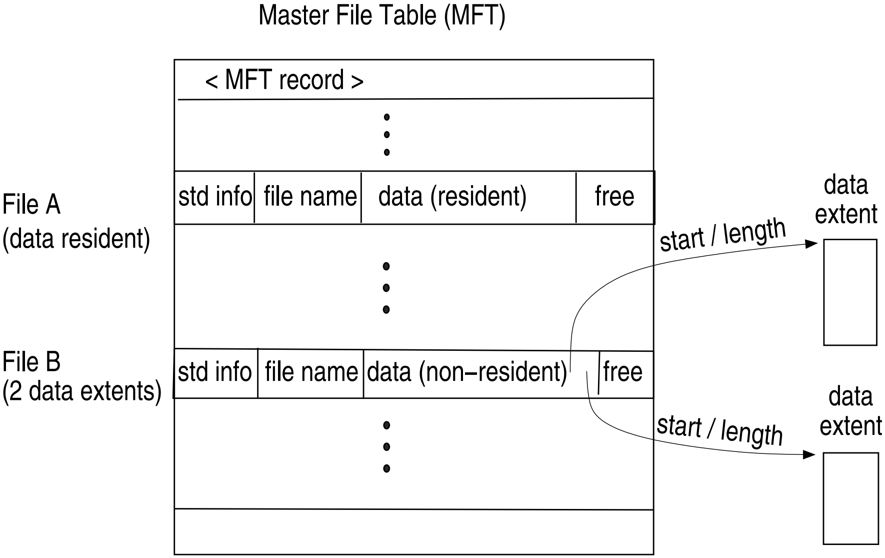
Outline

1. Filesystem Interface
2. Persistent Storage Devices: Hard disks
3. Filesystem Implementation
4. FAT Filesystem
5. FFS: Unix Fast Filesystem
6. NTFS: Microsoft Filesystem
7. Crash Consistency (OSTEP 42)
8. Log-structured Filesystems (OSTEP 43)
9. Persistent Storage Devices: Flash-based SSDs (OSTEP 44)

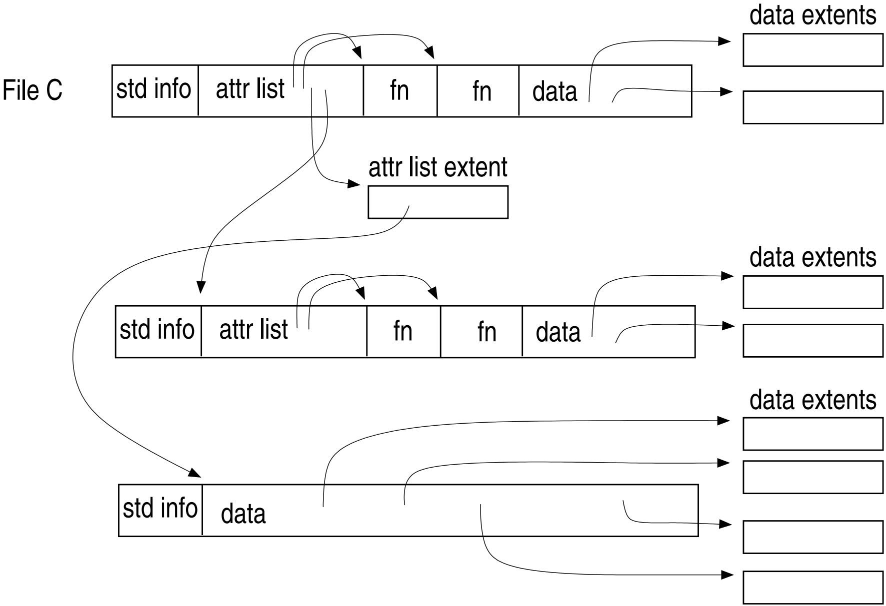
NTFS Index Structure

- Master File Table (MFT)
  - corresponds to FFS inode array
  - holds an array of 1KB MFT records
- MFT Record: sequence of variable-size attribute records
- Std info attribute: owner id, creation/mod/... times, security, ...
- File name attribute: file name and number
- Data attribute record
  - data itself (if small enough), or // resident
  - list of data "extents" (if not small enough) // non-resident
- Attribute list
  - pointers to attributes in this or other MFT records
  - pointers to attribute extents
  - needed if attributes do not fit in one MFT record
    - eg, highly-fragmented and/or highly-linked

Example: Files with single MFT record



Example: File with three MFT records



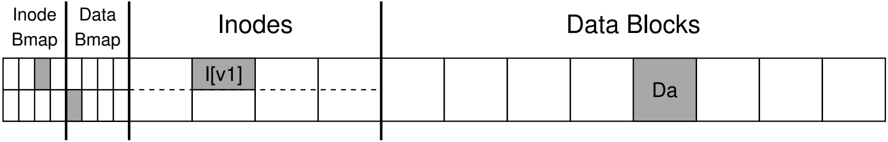
Outline

Crash Consistency

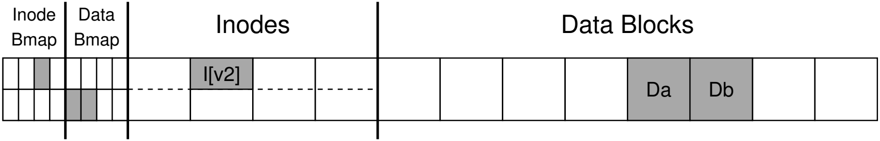
- Goal: to bring up a crashed fs in a consistent recent state
- Filesystem checker (fsck)
  - no extra work done during normal operation
  - recovery: examine fs metadata (bitmaps, inodes) to detect inconsistencies
  - too slow to handle disks of current size
- Journaling: Data or Metadata
  - do extra work during normal operation
  - recovery: check only the modified part

Example workload

- Fs: 8-bit inode bitmap, 8-bit data bitmap, 8 inodes, 8 data blocks



- Append of a single datablock to the existing file
  - requires 3 writes: data bitmap block, inode block, data block
  - write first to memory cache, later to disk



Crash scenarios

- Disk
  - writes blocks in arbitrary order
  - ensures a block write is either all or nothing
- Possible crash scenarios
  - 1 block written
    - Db or // ok
    - I[v2] or // inconsistency with data bitmap
    - B[v2] // space leak
  - 2 blocks written
    - [I[v2], B[v2]] or // bad. fs has garbage
    - [I[v2], Db] or // inconsistency with data bitmap
    - [B[v2], Db] // inconsistency with data bitmap

Filesystem checker

- Examines and, if needed, modifies to achieve consistency
  - Superblock // compare against duplicate superblocks
  - Inodes and indirect blocks to produce a correct version of bitmaps
  - Inode state
  - Inode links
  - Duplicates
  - Directory checks

Journaling

- Before overwriting disk structures in place, write a log in a specified place on disk

  Linux ext3 filesystem with a journal:

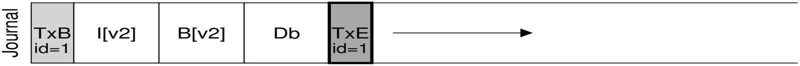
Super	Journal	Group 0	Group 1	. . .	Group N

- Crash during writing log:
  - at recovery: detect incomplete log, discard update
- Crash after writing log but before updating disk structure:
  - at recovery: replay log // (re)update disk structures

Data Journaling

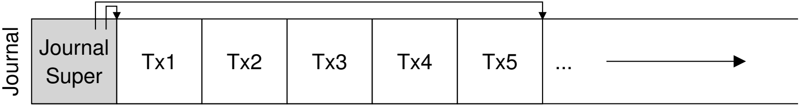
- Journal write

- Journal commit // start after journal write completes



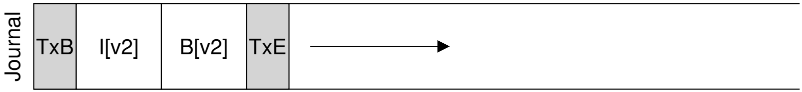
- Checkpoint: write log metadata & data to final on-disk locations // start after journal commit completes

- For better performance: batch log updates
- To keep the log bounded: circular log
  - journal superblock: stores start and end of log



Metadata Journaling

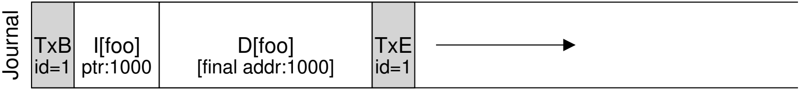
- Data write: to final on-disk locations
- Journal metadata write: write metadata into log
  // after above completes
- Journal commit // after above completes



- Checkpoint metadata: write metadata to final locations
  // after above completes
- Free: later, mark transaction free in journal superblock

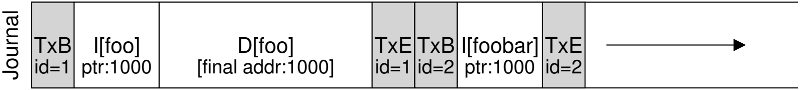
Tricky: metadata journaling and block reuse

- User adds an entry to directory foo (datablock 1000)



Note: directory data is treated as metadata

- User deletes directory foo and its contents
- User creates new file foobar which ends up using block 1000



- Crash occurs after log is complete
  - recovery action overwrites foobar data with foo data

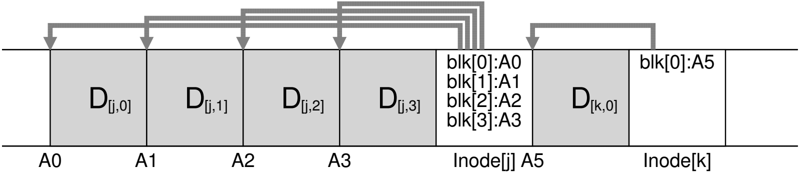
Outline

Motivation

- Memory size is growing rapidly
  - so fs performance determined by writes
- Disk: random io is much slower than sequential io
  - seek and rotational delays decreasing slowly
  - data bandwidth increasing rapidly
- Existing filesystems involve lots of random writes
  - create new 1-block file:
    - new inode, new datablock
    - bitmaps
    - parent inode, parent datablock(s)
- Existing filesystems are not RAID-aware
  - small-write problem

Basic idea

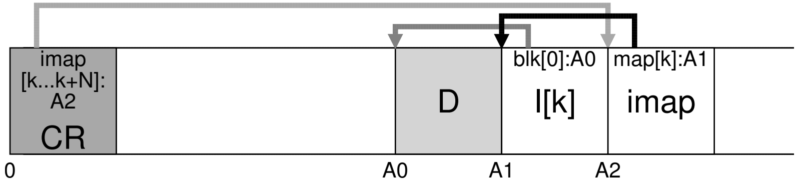
- Make the filesystem a log
- Do all writes (data + metadata) to a (large) in-memory segment
  - Eg: [4 block writes to file j], [1 block write to file k]



- When segment is full, write it to disk sequentially (not in-place)
- But now the on-disk inodes are not in fixed locations
  - need a dynamic map of on-disk locations of current inodes
- Also need to free segment-size areas on disk
  - garbage collect old inodes and datablocks
  - coalesce small holes to segment-sized hole

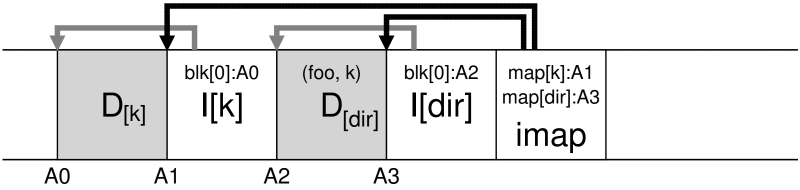
Solution to locating current inodes

- Inode map (imap): inode # $\longrightarrow$ disk location of "current" inode
- Imap on disk (for crash recovery) and in memory (for speed)
- On-disk imap in fixed place & frequently updated $\Rightarrow$ random io
- Instead on-disk imap is spread over log
  - upon an inode write, also write the relevant chunk of imap
  - checkpoint region (CR) gives locations of current imap chunks
  - CR at a fixed location in disk



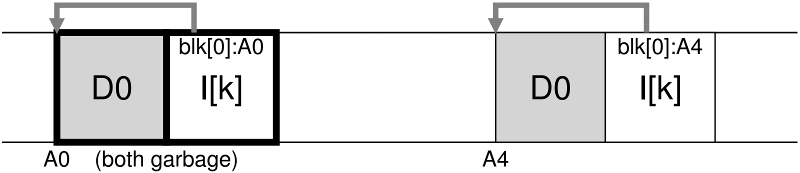
Directory update example

- Create file foo in a directory dir

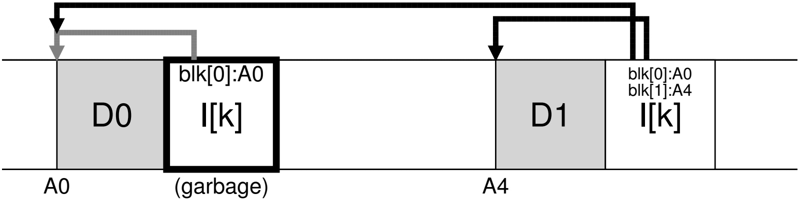


Garbage formation

- Example: given file k with 1 datablock, update the datablock



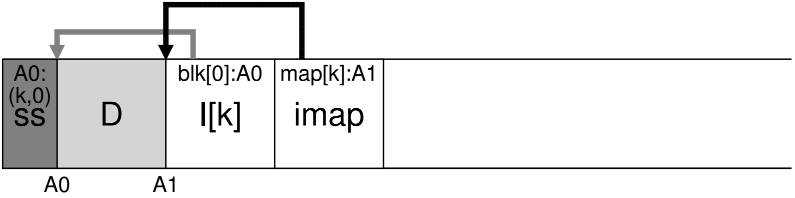
- Example: given file k with 1 datablock, append a new datablock



Garbage collection

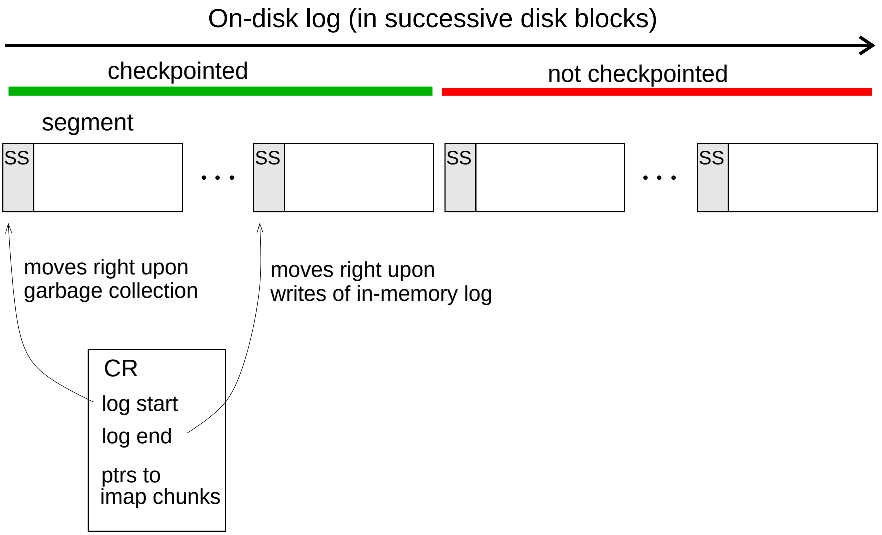
- LFS cleaner works segment by segment
  - read in M old segments
  - determine the live blocks in those segments
  - write them out (compactly) into N new segments (N < M)

- Determining block liveness
  - every segment X has a segment summary (SS) block
  - for each datablock D in X: disk location, inode number, offset



  - D is garbage if
    D.location in SS $\neq$ D.offset location in current inode

Invariant for on-disk content



Crash recovery

- Crash can occur while writing on-disk log or updating CR
- LFS updates the CR every 30 sec
- To ensure atomic update of CR
  - maintain two on-disk CRs and update them alternately
  - CR update
    - write timestamp in CR header, write CR
    - then write same timestamp in CR trailer
- Crash recovery:
  - choose the latest CR with consistent timestamps
  - from this CR, construct an in-memory imap
  - start logging at the CR's log-end position.
- But loses all of the uncheckpointed log (30 sec!)

Better crash recovery

- Store some additional info in normal operation
  - in segment summary (SS): location of every inode in the segment
  - each segment has matching timestamps at start and end
- Upon crash, recover to the last checkpoint (as before)
- Then roll forward on the uncheckpointed part of the log
  - while the next segment has matching timestamps:
    update the CR with the segment's metadata

Outline

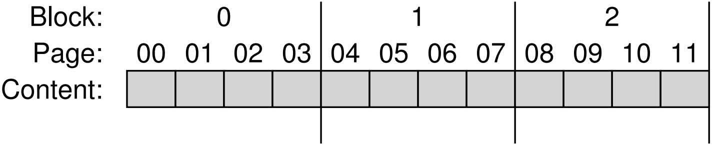
Flash Devices

- RAM with a floating insulated gate
- No moving parts
  - more robust than disk to impacts
  - consumes less energy
  - uniform access time // good for random access
- Wears out with repeated usage? Lose charge when unused?

- Read/write operations
  - done in blocks (12KB to 128KB)
  - tens of microseconds // if erased area available for writes
- Write requires erased area
- Erase operation
  - erasure block (few MB)
  - tens of milliseconds
  - so maintain a large erased area

Flash chip

- Consists of blocks divided into pages
  - blocks are large: 128 KB, 256 KB, ...
  - pages are reasonable: 4 KB, ...



- Page is invalid (i), erased (E) or valid (V)
  - any page can be read // $\approx 25\mu s$
  - an erased page can be written // $\approx 300\mu s$
  - only blocks can be erased, not individual pages // $\approx 2000\mu s$
- No moving parts: reliable
- Block wears out after repeated erase-write cycles

Raw Flash to Flash-based SSD

- Goal: provide an array of read-write pages
- Achieved via a flash translation layer (FTL)

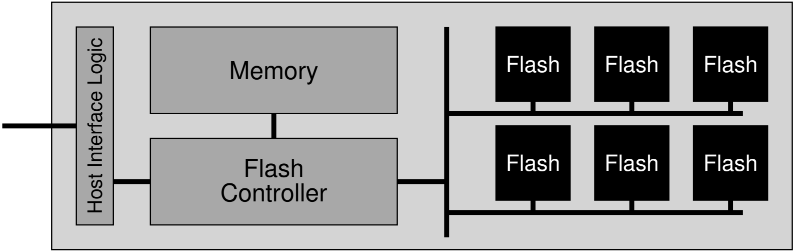


Figure 44.3: **A Flash-based SSD: Logical Diagram**

Log-structured FTL – 1

- Eg: write blocks 100, 101, 2000, 2001 with content a1, a2, b1, b2

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	i	i	i	i	i	i	i	i	i	i	i	i

Content:												
State:	E	E	E	E	i	i	i	i	i	i	i	i

Table: 100 → 0 101 → 1 2000 → 2 2001 → 3 Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a1	a2	b1	b2									Chip
State:	V	V	V	V	i	i	i	i	i	i	i	i	

Log-structured FTL – 2

- Garbage collection
  - suppose user overwrites user blocks in pages x_0, x_1, $\cdots$
  - new content goes into new (erased) pages y_0, y_1, $\cdots$
  - map is updated
  - old pages are garbage collected
    - live content in old blocks are moved to new blocks
    - block of old pages is erased
- Hybrid mapping
  - page map only would become too large
  - also use block map
    - treat user block n as $[n_1, n_2]$, where n_1 maps to a flash block
    - user blocks $[n_1, 0]$, $[n_1, 1]$, $\cdots$ in pages 0, 1, $\cdots$ of a block
- Map stored in flash // as in log-structured filesystems
  - at unmount for persistence
  - periodically for crash recovery

Hybrid mapping example – 1

- Suppose blocks 1000–1003 are stored in pages 8–11

Log Table:

Data Table: 250 → 8

Memory

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:									a	b	c	d
State:	i	i	i	i	i	i	i	i	V	V	V	V

Flash Chip

- Suppose user overwrites these blocks with new content

Log Table: 1000 → 0 1001 → 1 1002 → 2 1003 → 3

Data Table: 250 → 8

Memory

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a’	b’	c’	d’					a	b	c	d
State:	V	V	V	V	i	i	i	i	V	V	V	V

Flash Chip

Hybrid mapping example – 2

- Garbage collection then yields

Log Table:
Data Table: 250 → 0

Memory

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a'	b'	c'	d'								
State:	V	V	V	V	i	i	i	i	i	i	i	i

Flash Chip

SSDs vs HDDs

- Performance

Device	Random Reads (MB/s)	Random Writes (MB/s)	Sequential Reads (MB/s)	Sequential Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Figure 44.4: **SSDs And Hard Drives: Performance Comparison**

- Cost: HDDs much cheaper than SSDs (eg, 10 times less)