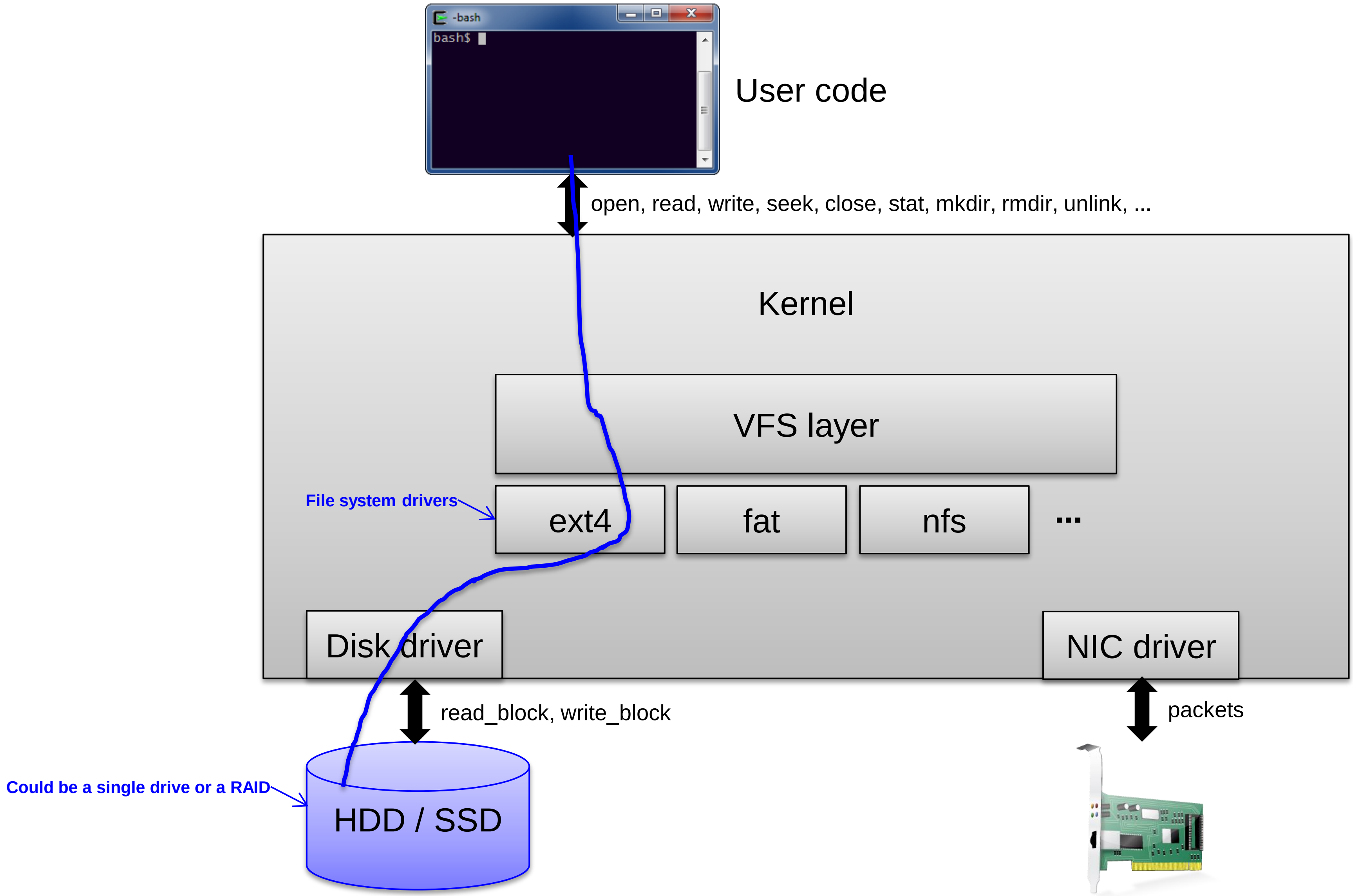
# ECE590-03
# Enterprise Storage Architecture

# Fall 2016

## File Systems

Tyler Bletsch
Duke University

# The file system layer



# Disk file systems

- All have same goal:
  - Fulfill file system calls (open, seek, read, write, close, mkdir, etc.)
  - Store resulting data on a block device
- The big (non-academic) file systems
  - **FAT ("File Allocation Table")**: Primitive Microsoft filesystem for use on floppy disks and later adapted to hard drives
    - FAT32 (1996) still in use (default file system for USB sticks, SD cards, etc.)
    - Bad performance, poor recoverability on crash, but near-universal and easy for simple systems to implement
  - **ext2, ext3, ext4**: Popular Linux file system.
    - Ext2 (1993) has **inode**-based on-disk layout – much better scalability than FAT
    - Ext3 (2001) adds **journaling** – much better recoverability than FAT
    - Ext4 (2008) adds various smaller benefits
  - **NTFS**: Current Microsoft filesystem (1993).
    - Like ext3, adds **journaling** to provide better recoverability than FAT
    - More expressive metadata (e.g. Access Control Lists (ACLs))
  - **HFS+**: Current Mac filesystem (1998). Probably good I guess?
  - "Next gen" file systems: **ZFS** (2005)**, btrfs** (2009)**, WAFL** (1998), and others
    - Block indirection allows snapshots, copy-on-write clones, and deduplication
    - Often, file system handles redundancy itself – no separate RAID layer

# FAT

# FAT

- FAT: "File Allocation Table"
- 3 different varieties, FAT12, FAT16, FAT32 in order to accommodate growing disk capacity

- Allocates by **clusters** (a set of contiguous disk sectors)
  - Clusters number is a power of two < $2^{16}$

- The actual File Allocation Table (FAT):
  - Resides at the beginning of the volume
  - Two copies of the table
  - For a given cluster, gives next cluster (or FFFF if last)

| | | | | |
|---|---|---|---|---|
| Partition Boot Sector | FAT1 | FAT2 (duplicate) | Root folder | Other folders and all files. |



Adapted from "Computer Forensics: Hard Drive Format" by Thomas Schwarz (Santa Clara Univ)

# Directories

- Root directory:
  - A fixed length file (in FAT16, FAT32)
- Subdirectories are files of same format, but arbitrary size (extend via the FAT)
- Consist of 32B entries:

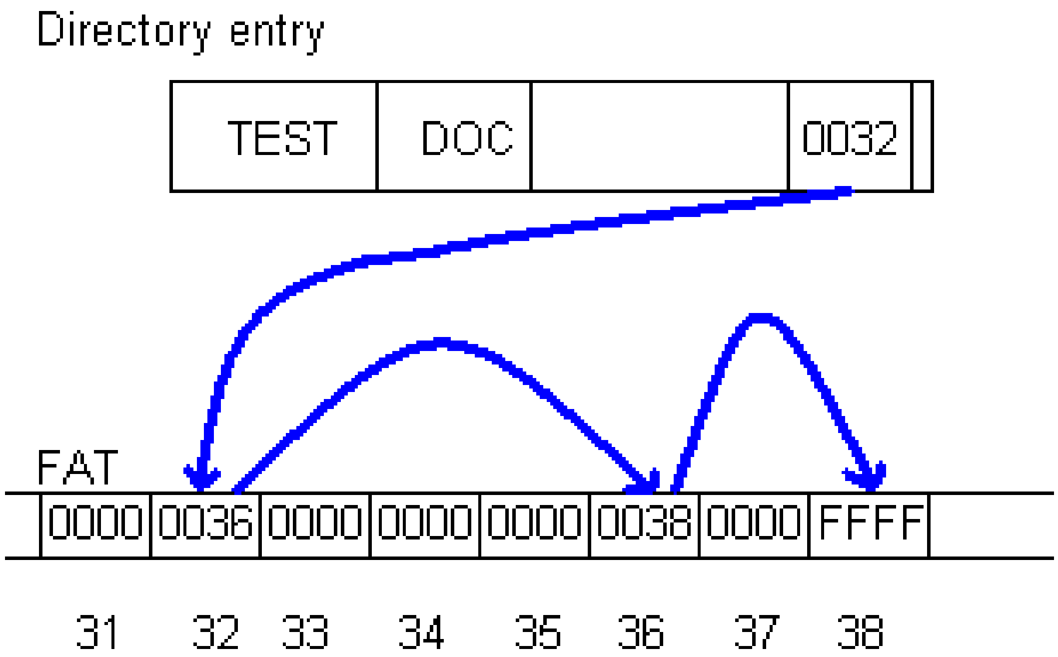
| Offset | Length | Meaning |
|---|---|---|
| 0x00 | 8B | File Name |
| 0x08 | 3B | Extension |
| 0x0b | 1B | File Attribute |
| 0x0c | 10B | Reserved:<br>(Create time, date, access date in FAT 32) |
| 0x16 | 2B | Time of last change |
| 0x18 | 2B | Date of last change |
| 0x1a | 2B | First cluster |
| 0x1c | 4B | File size. |



Adapted from “Computer Forensics: Hard Drive Format” by Thomas Schwarz (Santa Clara Univ)

# FAT Principle

- Directory gives first cluster
- FAT gives subsequent ones in a simple table
- Use FFFF to mark end of file.






Adapted from “Computer Forensics: Hard Drive Format” by Thomas Schwarz (Santa Clara Univ)

# Tradeoffs

- Cluster size
  - Large clusters waste disk space because only a single file can live in a cluster.
  - Small clusters make it hard to allocate clusters to files contiguously and lead to large FAT.

- FAT entry size
  - To save space, limit size of entry, but that limits total number of clusters.
  - FAT 12: 12 bit FAT entries
  - FAT 16: 16 bit FAT entries
  - FAT 32: 32 bit FAT entries

# Long file names

- Needed to add support for filenames longer than 8+3
- Also needed to be backward compatible

- Result: ridiculous but it works
  - Store a bunch of extra “invalid” entries after the normal one just to hold the long file name
  - Set up these entries in such a way that old software will just ignore them
  - Every file has a long name and a short (8+3) name; short name is auto-generated


Adapted from “Computer Forensics: Hard Drive Format” by Thomas Schwarz (Santa Clara Univ)

# Problems with FAT

1. **Scalability/efficiency:**
   - Every file uses at least one cluster: internal fragmentation
   - No mechanism to optimize data locality (to reduce seeks): external fragmentation
   - Fixed size FAT entries mean that larger devices need larger clusters; problem gets worse
2. **Consistency**: What happens when system crashes/fails during a write? Nothing good...
3. **Like a billion other things**: Seriously, did you see the long filename support? It's awful. And there is literally no security model – no permissions or anything. There's just a "hidden" bit (don't show this unless the user really wants to see it) and a "system" bit (probably don't delete this but you can if you want to). It's impossible to support any kind of multi-user system on FAT, so Windows basically didn't until NT, which didn't become mainstream until Windows 2000 and later XP. Also, the way you labeled a whole file system was a special file that had a special permission bit set – that's right, there's a permission bit for "this file is not really a file but rather the name of the file system". Also, the directory entries literally contain a "." entry for the current directory, which is completely redundant. Speaking of redundant data, the duplicate FAT has no parity or error recovery, so it only helps you if the hard drive explicitly fails to read a FAT entry, not if there's a bit error in data read. Even so, if the disk does fail to read the first FAT, the second only helps if the duplicate has the entry you need intact. But recall that bad sectors tend to be clustered, so a failure of one part of the FAT usually means the whole FAT region is dead/dying. This meant scores of FAT data was lost to relatively small corruptions, because file recovery is almost impossible if all disk structure information is lost. In any case, we haven't even got to the other backwards compatibility stuff in FAT32. In that format, the bytes that make up the cluster number aren't even contiguous! They sacrified some of the reserved region, so just to compute the cluster number you have to OR together two fields. Worst thing of all is that despite all this, FAT32 is still alive and well with no signs of going away, because it's so common that every OS supports it and it's so simple that cheap embedded hardware can write to it. We live in a nightmare.

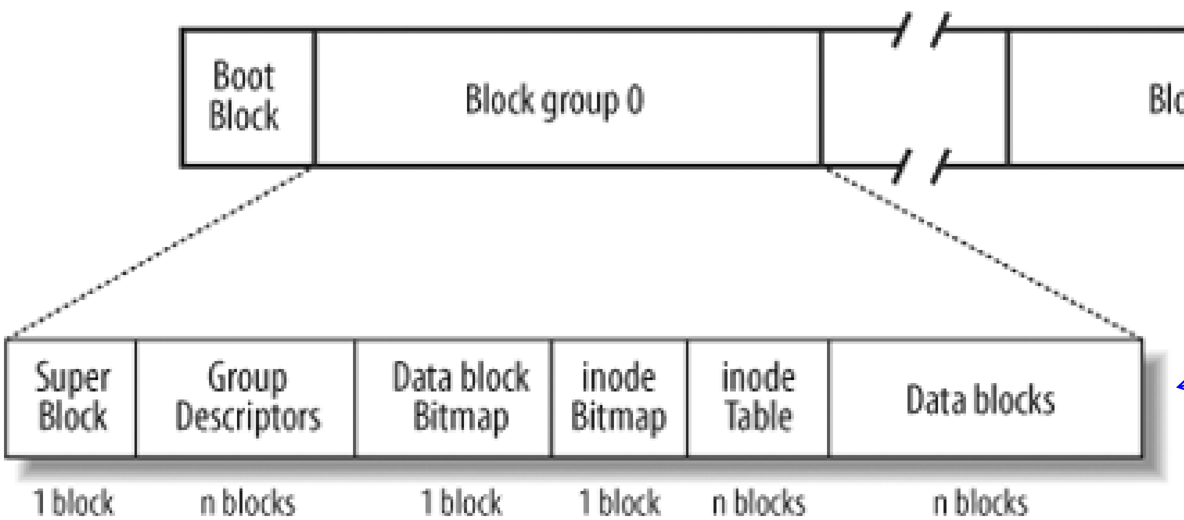
# ext2

# Disk Blocks

- Allocation of disk space to files is done with blocks.
- Choice of block size is fundamental
  - Block size small: Needs to store much location information
  - Block size large: Disk capacity wasted in partially used blocks (at the end of file)
- Typical Unix block sizes are 4KB and 8KB

# Disk layout

- **Super block**: Filesystem-wide info (replicated a lot)
- **Group descriptors**: addresses of the other parts, etc.
- **Data block bitmap**: which blocks are free?
- **Inode bitmap**: which inodes are free?
- **Inode table**: the inodes themselves
- **Data blocks**: actual file data blocks




Original UNIX filesystem basically had one of this for the whole disk, which meant that metadata was always really far from data. This more modern "block group" idea drastically reduces the average distance between the two.



From "Understanding the Linux Kernel, 3e" by Marco Cesati, Daniel P. Bovet.

# Inodes

- Inodes are fixed sized metadata describing the layout of a file
- Inode structure:
  - i_mode (directory IFDIR, block special file (IFBLK), character special file (IFCHR), or regular file (IFREG)
  - i_nlink
  - i_addr (an array that holds addresses of blocks)
  - i_size (file size in bytes)
  - i_uid (user id)
  - i_gid (group id)
  - i_mtime (modification time & date)
  - i_atime (access time & date)

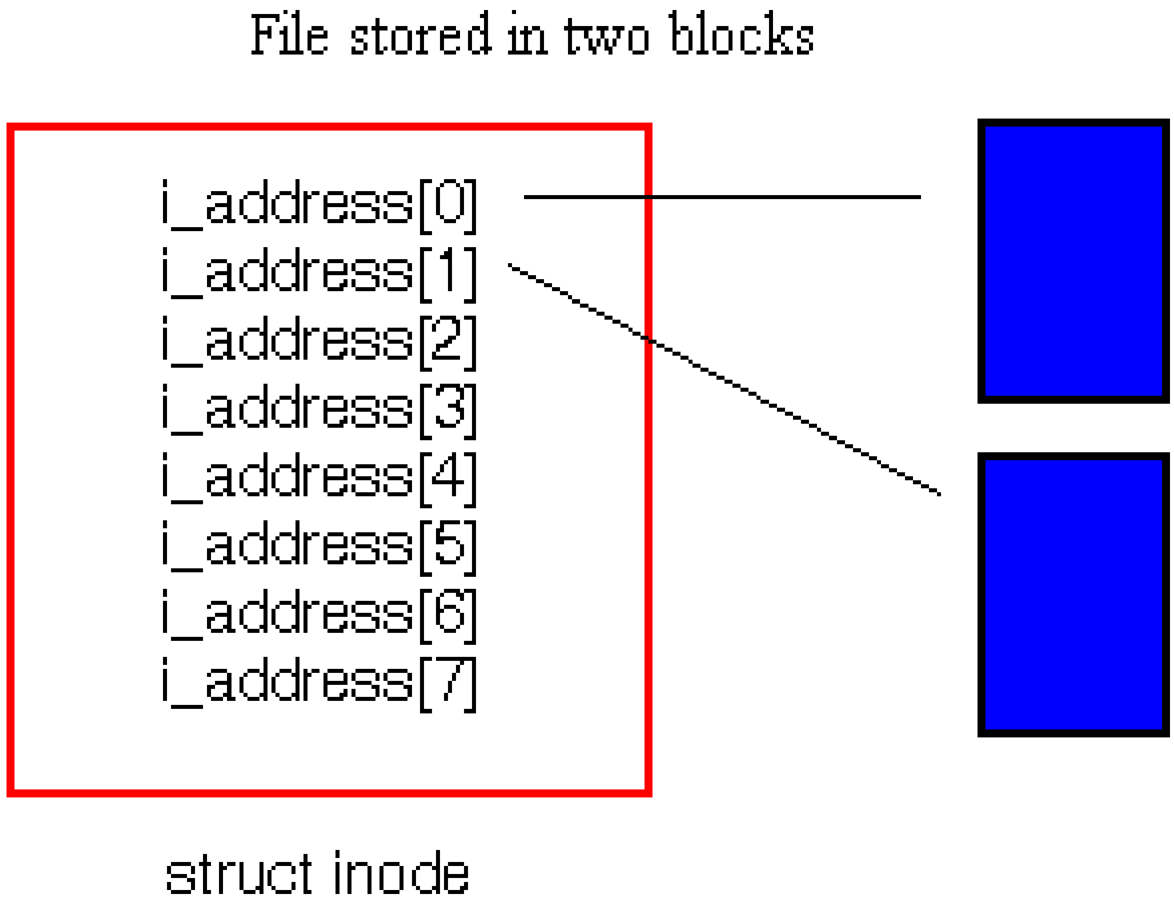
# Inodes

- Metadata in Inode is space-limited
  - Limited NUMBER of inodes:
    - Inode storing region of disk is fixed when the file system is created
    - Run out of inodes -> can't store more files -> Can get "out of disk" error even when capacity is available
  - Limited SIZE of inode:
    - Number of block addresses in a single inode only suffices for small files
    - Use (single and double) indirect inodes to find space for all blocks in a file



Adapted from "Computer Forensics: Unix File Systems" by Thomas Schwarz (Santa Clara Univ)

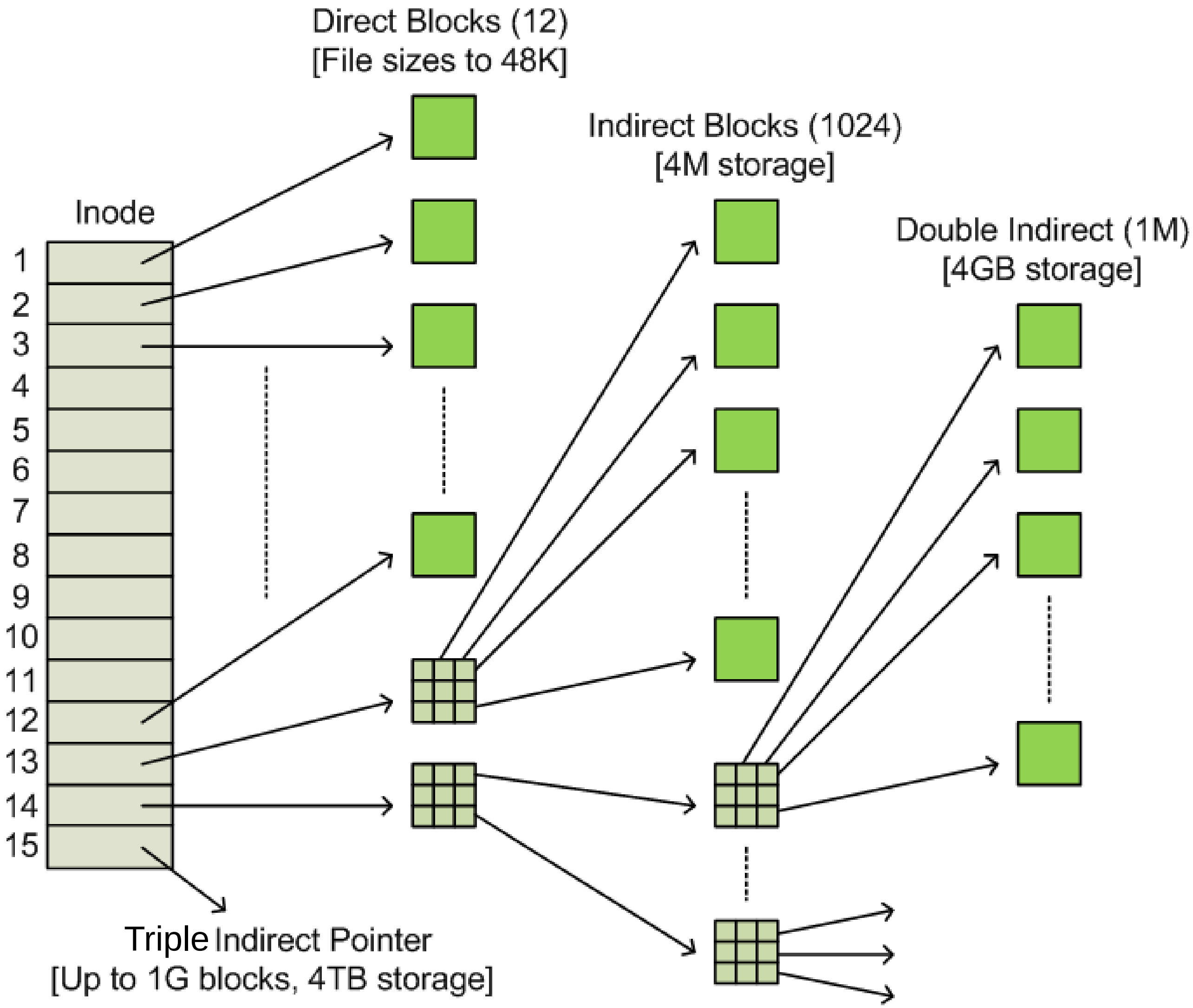
# Inode indirection






Adapted from “Computer Forensics: Unix File Systems” by Thomas Schwarz (Santa Clara Univ)

# Inode indirection




From "File Systems Indirect Blocks and Extents" by Cory Xie ([link](link))

# Directories and hard links

- Directories are special files that list file names and inode numbers
(and some other minor metadata)

- What if two directories refer to the same inode number?
  - Two "files" that are actually the same content
  - This is called a **hard link**
  - Need to track "number of links" – deallocate inode when zero
  - This is an early example of filesystem-based storage efficiency:
    - Can store same data "twice" without actually storing more data!
    - Example: "Rsnapshot" tool can create multiple point-in-time backups while eliminating redundancy in unchanged files
    - *We'll see more advanced forms of filesystem-based storage efficiency later on!*

# EXT Allocation Algorithms

- Allocation – selecting block group:
  - Non-directories are allocated in the same block group as parent directory, if possible.
  - Directory entries are put into underutilized groups.

- Deallocation - deleted files have their inode link value decremented.
  - If the link value is zero, then it is unallocated.

# Soft links

- **Soft link**: an additional file/directory name.
  - Also called **symbolic link** or **symlink**.
  - A special file whose **contents** is the **path** to another file/directory.
  - Path can be relative or absolute
  - Can traverse file systems
  - Can point to nonexistent things
- Can be used as file system organization "duct tape"
  - Organize lots of file systems in one place (e.g., cheap NAS namespace virtualization)
  - Symlink a long, complex path to a simpler place, e.g.:

```
$ ln -s /remote/codebase/projectX/beta/current/build ~/mybuild
$ cd ~/mybuild
```

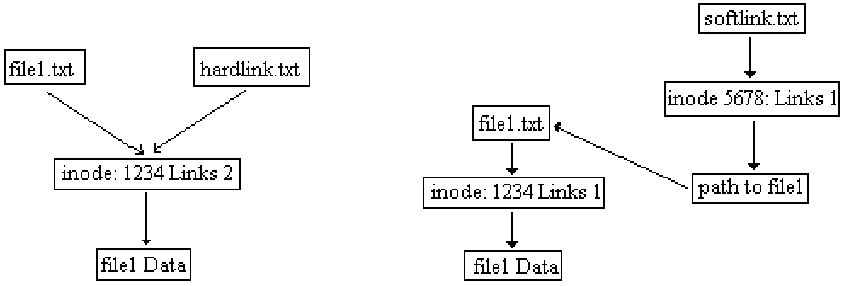



Figure from "Computer Forensics: Unix File Systems" by Thomas Schwarz (Santa Clara Univ)

# EXT Details: Two time issues

- Time Values
  - Are stored as seconds since January 1, 1970, Universal Standard Time
  - Stored as 32-bit integer in most implementations
  - Remember Y2k? Get ready for the Year 2038 problem.
- Linux updates (in general)
  - A-time, when the content of file / directory is read.
  - This can be very bad: every read implies a write!!
  - Can be disabled: “noatime” option (atime field becomes useless)
  - Can be mitigated: “relatime” option – only update atime if file modified since current atime or if atime difference is large

# Problems with ext2

- We solved the scalability/efficiency problem from FAT

- We still have one big problem left:

  **Consistency**: What happens when system crashes/fails during a write? Nothing good...

# Journaling: ext3, NTFS, and others

# Why Journaling?

- Problem: Data can be inconsistent on disk
  - Writes can be committed out of order
  - Multiple writes to disk need to all occur and “match” (e.g. metadata of file size, inode listing of disk blocks, actual data blocks)

- How to solve?
  - Write our *intent* to disk ahead of the actual writes
  - These “intent” writes can be fast, as they can be ganged together (few seeks)
  - This is called **journaling**

# Design questions

- Where is journal?
  - Same drive, separate drive/array, battery backed RAM, etc.

- What to journal?
  - Logical journal
    - Metadata journaling: Only log meta data in advance
  - Physical journal
    - Data journaling: Log advanced copy of the data
      (All data written twice!)

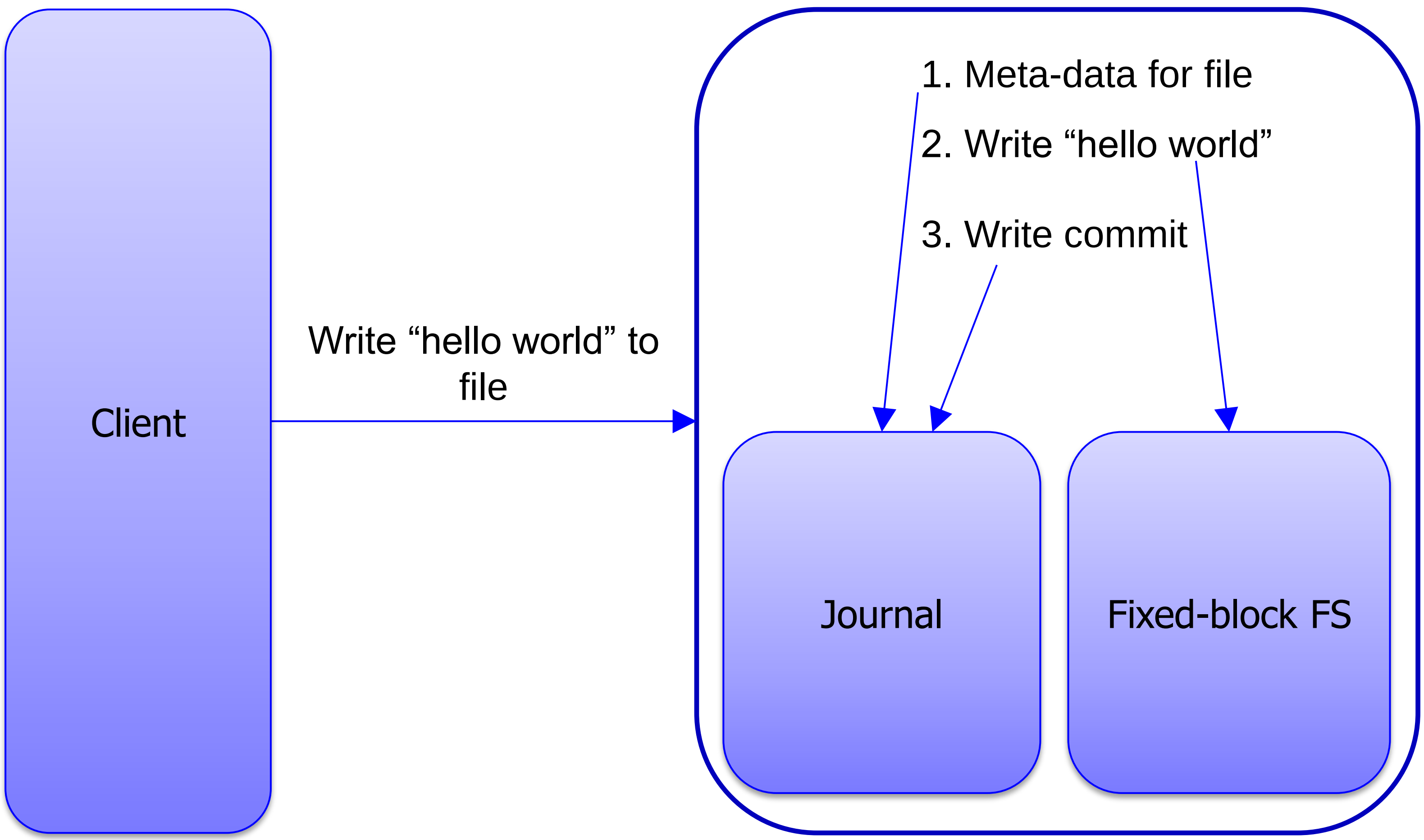
- What are the tradeoffs?
  - Costs vs. benefits

From "Journaling Filesystems" by Vince Freeh (NCSU)

# Journaling

- Process:
  - record changes to cached metadata blocks in journal
  - periodically write the journal to disk
  - on-disk journal records changes in metadata blocks that have not yet themselves been written to disk
- Recovery:
  - apply to disk changes recorded in on-disk journal
  - resume use of file system
- On-disk journal: two choices
  - maintained on same file system as metadata, OR
  - stored on separate, stand-alone file system

# Journaling File System




From "Analysis and Evolution of Journaling File Systems" by Vijayan Prabhakaran, Andrea and Remzi Arpai-Dusseau, and Andrew Quinn (Univ. Michigan), 2016

# Journaling Transaction Structure

- A journal transaction
  - consists of all metadata updates related to a single operation
  - transaction order must obey constraints implied by operations
  - the memory journal is a single, merged transaction

- Examples
  - Creating a file
    - creating a directory entry (modifying a directory block),
    - allocating an inode (modifying the inode bitmap),
    - initializing the inode (modifying an inode block)
  - Writing to a file
    - updating the file's write timestamp ( modifying an inode block)
    - may also cause changes to inode mapping information and block bitmap if new data blocks are allocated



From "Operating Systems: File systems" by Dennis Kafura (Virginia Tech)

# Journaling in Linux (ext3)

- Given the (merged) transaction from memory
- Start flushing the transaction to disk
  - Full metadata block is written to journal
  - Descriptor blocks are written that give the home disk location for each metadata block
- Wait for all outstanding filesystem operations in this transaction to complete
- Wait for all outstanding transaction updates to be completely
- Update the journal header blocks to record the new head/tail
- When all metadata blocks have been written to their home disk location, write a new set of journal header blocks to free the journal space occupied by the (now completed) transaction

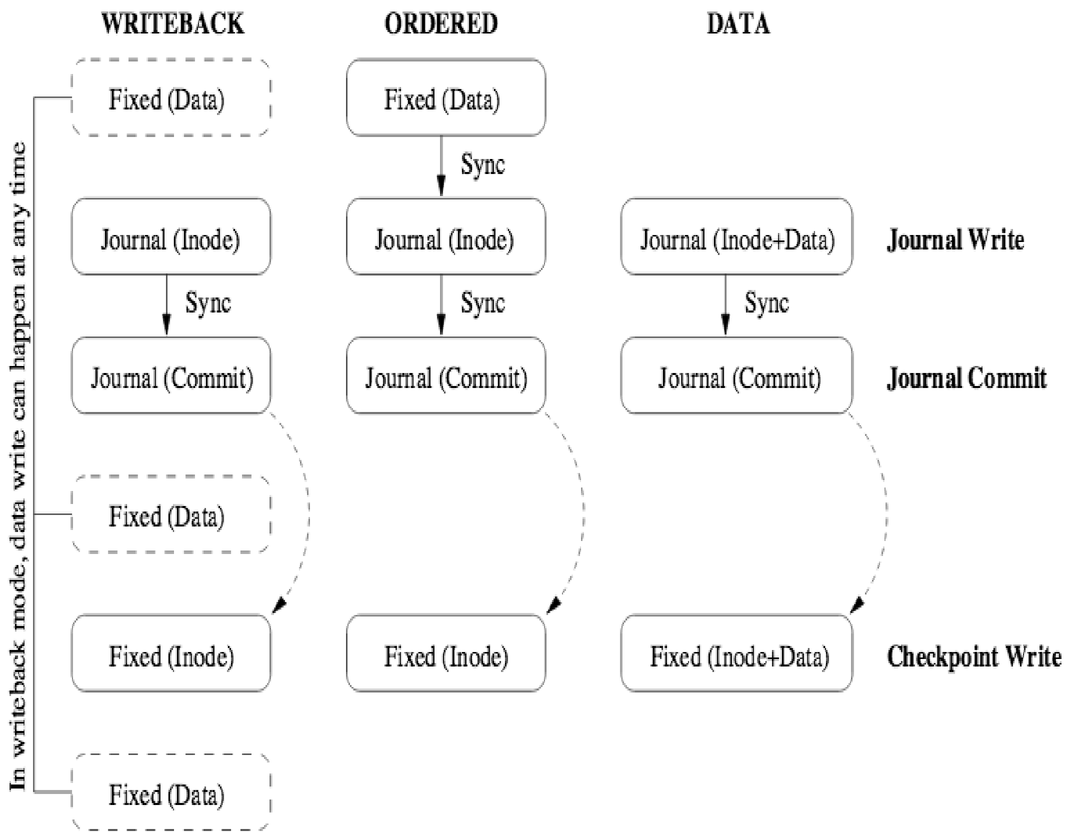


From “Operating Systems: File systems” by Dennis Kafura (Virginia Tech)

# Journaling modes (ext3)

1. **Write-back**: meta-data journaled – no enforced ordering between fixed location data and journal writes. *only guarantees meta-data crash consistency*
2. **Ordered**: meta-data journaled – enforces that data is written out before journal commit. *guarantees consistency recovery*
3. **Data-journaling** mode: meta-data and data are journaled: typically writes data twice!

- **Check-pointing**: writing the journaled meta-data/data to the fixed - locations






From "Analysis and Evolution of Journaling File Systems" by Vijayan Prabhakaran, Andrea and Remzi Arpai-Dusseau, and Andrew Quinn (Univ. Michigan), 2016

# Who does journaling?

- Everyone does journaling.
  - Microsoft Windows: NTFS
  - Linux: ext3, ext4, jfs, reiserfs
  - Apple OSX: HFS+
- Full list:

| | |
|---|---|
| GFS | OCFS |
| GPFS | OCFS2 |
| HPFS | XFS |
| NTFS | JFS |
| HFS | QFS |
| HFS Plusline | Be File |
| FFS | NSS |
| UFS1 | NWFS |
| UFS2 | ODS-2 |
| LFS | ODS-5 |
| ext2 | UDF |
| ext3 | VxFS |
| ext4 | Fossil |
| Lustre | ZFS |
| NILFS | VMFS2 |
| ReiserFS | VMFS3 |
| Reiser4 | Btrfs |

# Can we go further?

- If journaling is so great, what if we just NEVER wrote to fixed blocks, and used the journal for EVERYTHING????

# Can we go further?

- Yes!

# Log-structured file systems

# Why LFS?

- CPU speed increasing faster than disk access is decreasing
  - What is impact of this?
- Read will be satisfied by cache (?)
  - Read performance does not matter
    - According to authors
  - Disk accesses are mostly writes
  - Optimize for the common case
- Benefits of LFS
  - Faster write performance
  - Same read performance (?)
  - Faster crash recovery

M. Rosenblum and J. K. Ousterhout. The design and implementation of a Log-structured File system. ACM TOCS, 10(1):26–52, 1992.

# Existing systems

- Four observations
  - Processor speeds are up
  - Disk seek time is not improving fast enough
  - Main memory & cache sizes are growing
  - Number of processors is increasing
- Workloads – what kinds, how to model
  - Different loads
  - Most difficult (for performance) is office load
    - Small files
    - Random disk I/O
    - Much creation/deletion → access to metadata
  - Regular, predictable workloads are not interesting

From “Journaling Filesystems” by Vince Freeh (NCSU)

# Two general problems

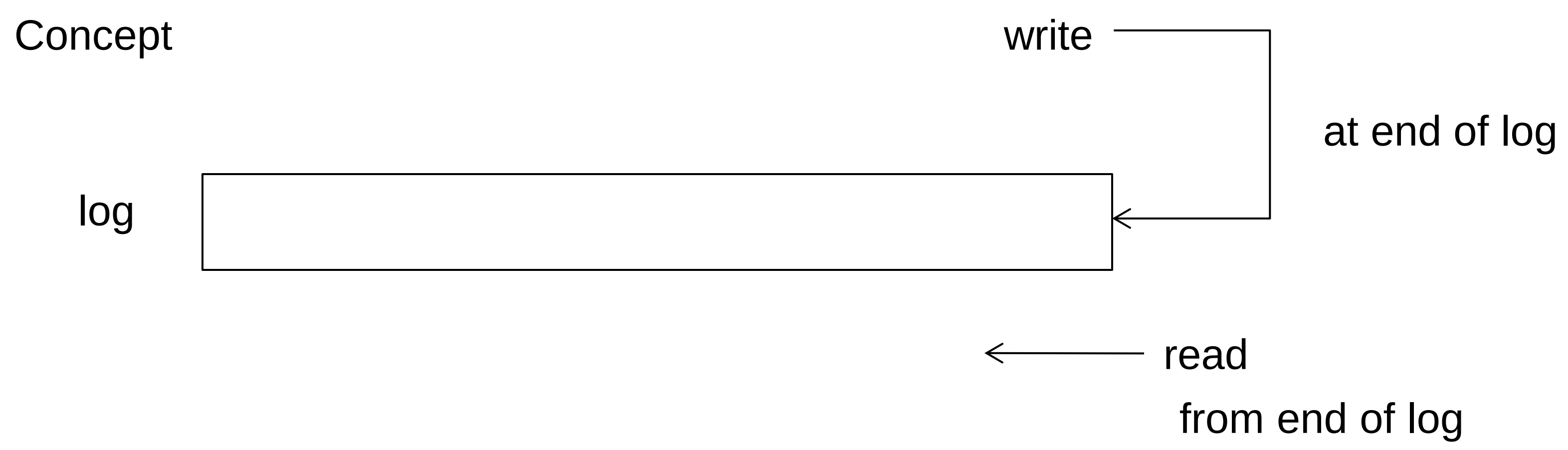
- Information is spread around
  - Many small accesses
  - Why is this bad?
  - How it is happening
    - Eg, 5 I/O to create file in FFS (predecessor to ext2)
- Synchronous writes
  - What: process waits for write to complete
  - Why: consistency
  - Why is it a problem
    - Process runs at disk speed
    - Does not benefit from CPU/memory increases
    - Poor write performance
    - Getting worse (relatively)

From “Journaling Filesystems” by Vince Freeh (NCSU)

# Key to LFS

- How does LFS achieve high write bandwidth?
  - Bundling writes
  - That's it…that's the whole idea of LFS
- How
  - Delay writes
  - Write large contiguous extents
- Key implementation issues
  - Retriving information from log
  - Managing free space

# Log-structured file system




More recently written block renders obsolete a version of that block written earlier.

| Issue | Approach |
|---|---|
| How to structure data/metadata | segments |
| How to manage disk space | segment cleaning |

# File location and reading

- Goal: match read performance of Unix (why match?)
- How is it done:
    - Inodes written to log
    - Inode location stored in inode map
- Keys
    - Inode is not at fixed location
    - Inode map is cached

# Free space

- Goal – maintain large free extents
- Circular log
  - Fill in
  - When get to end, go to beginning
  - If no room on disk, you're done (same as any FS)
  - Problem – fragmentation due to long-lived blocks



From "Journaling Filesystems" by Vince Freeh (NCSU)

# Solution space

- **Link** live blocks
  - Blocks are static
  - Problem
    - Over time it will be fragmented
    - Will not be different from FFS

- **Copying & compacting**
  - Move long-lived files to head of log
  - Compact the log
  - Problem
    - Too much copying

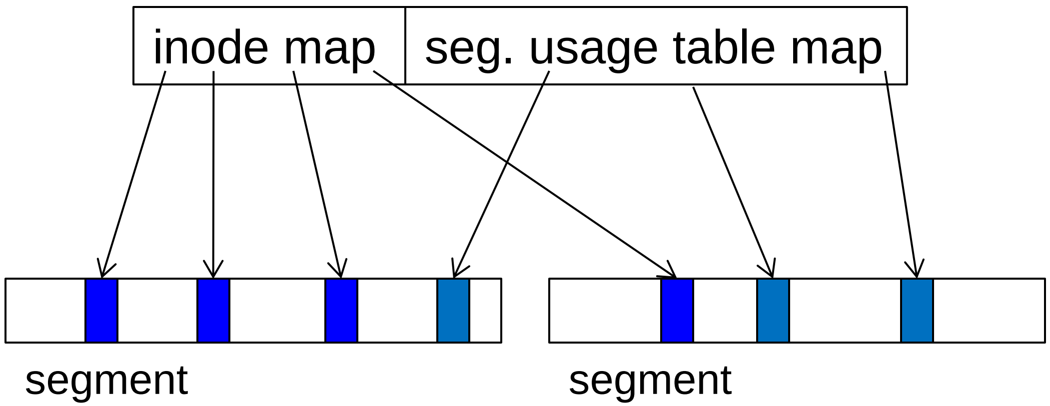
# Solution: Segments

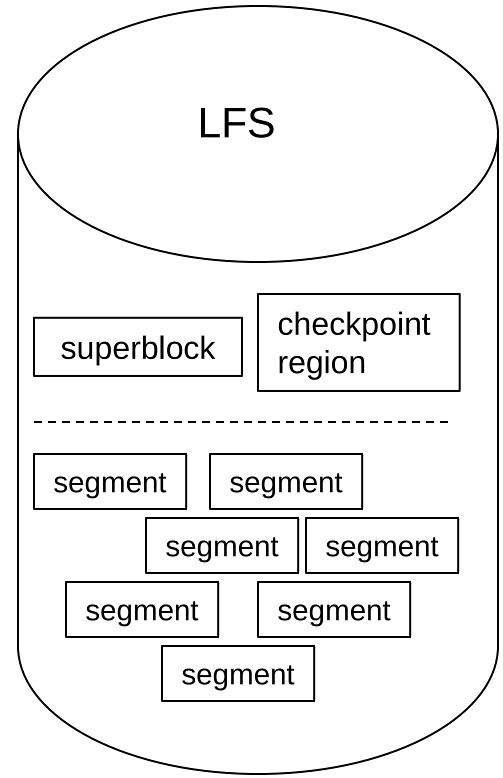
- **Segments**: a level of indirection
- A combination of linking and copying/compacting
- Compaction is confined to a segment
- How big should a segment be?

# LFS structure

Superblock - list: (segment, size)

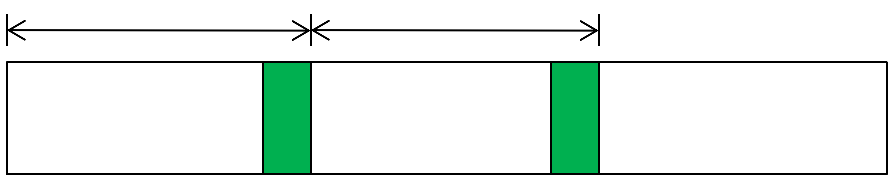
Checkpoint region:




inode map – list: (inode location, version#)
segment usage table – list: (live bytes, modified time)

segment



Segment summary block – list: (inode, version, block)

From “Operating Systems: File systems” by Dennis Kafura (Virginia Tech)

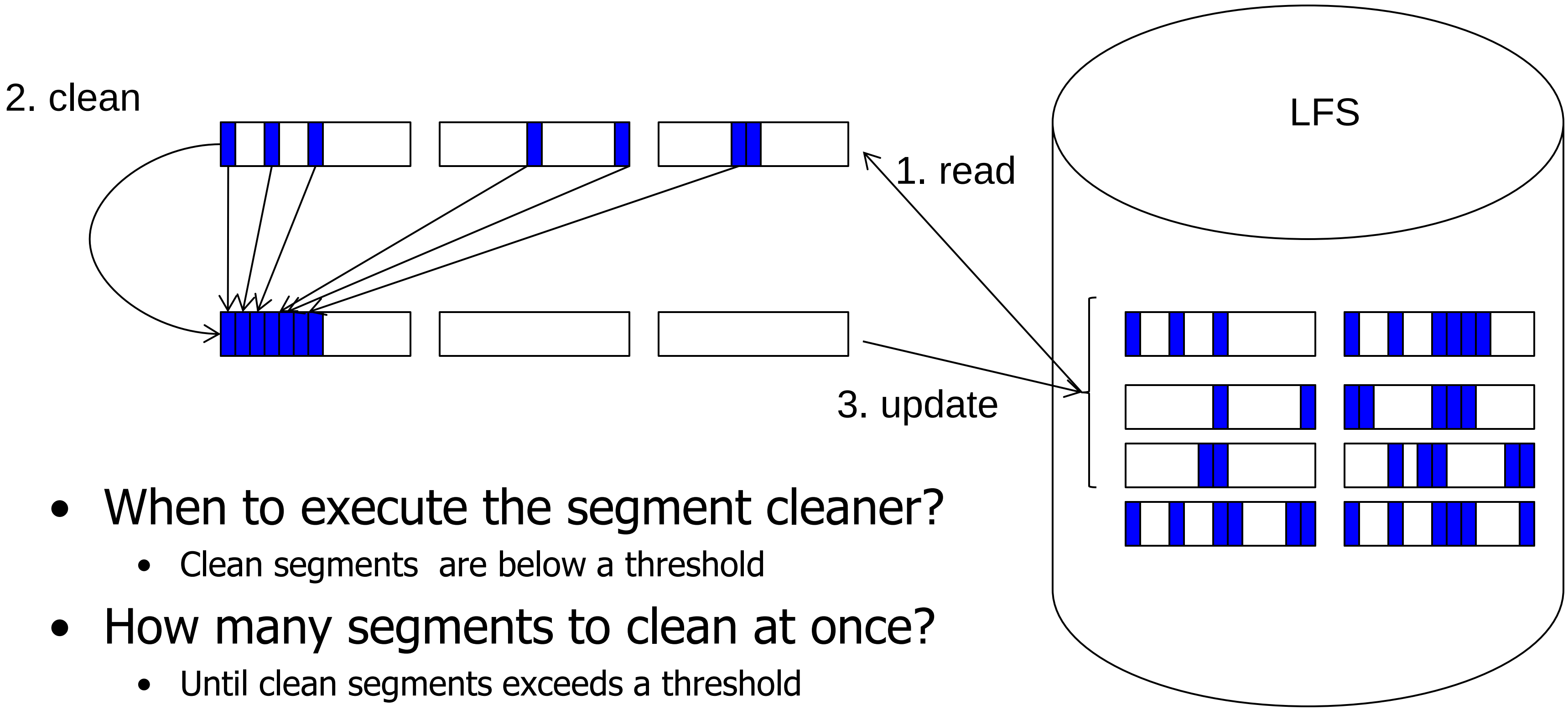
# Segment cleaning

- 3-step process
  - Read a number of segments into main memory
  - Find live data
  - Write back live data, re-claim segments
    - Problems: Uses cache, locks FS
- Segment summary block
  - Identifies live data
  - Eliminates need for freelists



From “Journaling Filesystems” by Vince Freeh (NCSU)

# Segment cleaning




- When to execute the segment cleaner?
  - Clean segments are below a threshold
- How many segments to clean at once?
  - Until clean segments exceeds a threshold
- Which segments to clean?
- How should live blocks be grouped?



From “Operating Systems: File systems” by Dennis Kafura (Virginia Tech)

# Use of log-structured filesystems

- In the role of a traditional filesystem – not a lot:
  - Original Ousterhout & Rosemblum LFS in Sprite OS (1992)
  - Various academic projects, some small commercial ventures
  - The NetApp "Write Anywhere File Layout (WAFL)"
    (we'll cover this one next)

- Specific to flash or optical media – more common
  (recall that those mediums have trouble with in-place writes):
  - UDF (commonly used on CD/DVD)
  - JFFS, JFFS2 (commonly used in for flash in embedded Linux systems)
  - Others (mostly focused around flash)

Note: "flash" above means raw flash, not SSDs – the data-hiding, wear-leveling, etc. done by SSDs obviates many of the benefits

# Remaining problem

- We've solved performance/efficiency issues with inodes and chunks (ext2)
- We've solved consistency with journaling (and perhaps logging)
- Remaining problem:
  - **Lack of magical superpowers that make you millions of dollars**

# Highly indirected filesystems

# Desires

- We want **snapshots**: point-in-time read-only replicas of current data which can be taken in O(1) time and space
- We want **clones**: point-in-time writable replicas of current data which can be taken in O(1) time and space, and we only store changes between clone and original
- We want various other features, like:
  - Directory-level **quotas** (capacity limits),
  - **Deduplication** (identify redundant data and store it just once), and
  - **Thin-provisioning** (provide storage volumes with a total capacity greater than actual disk storage available)

# Write Anywhere File Layout (WAFL)

- Copy-on-Write File System
- Inspired ZFS, HAMMER, btrfs
- Core Idea: Write whole snapshots to disk
- Snapshots are virtually free!
- Snapshots accessible from .snap directory in root

NetApp™

```
spike% ls -lut .snapshot/*/todo
-rw-r--r-- 1 hitz  52880 Oct 15 00:00
.snapshot/nightly.0/todo
-rw-r--r-- 1 hitz  52880 Oct 14 19:00
.snapshot/hourly.0/todo
-rw-r--r-- 1 hitz  52829 Oct 14 15:00
.snapshot/hourly.1/todo
```

From “Advanced File Systems” by Ali Jose Mashtizadeh (Stanford)

# File System Design for an NFS File Server Appliance

Dave Hitz, James Lau, and Michael Malcolm

**Technical Report TR3002**

**NetApp**

**2002**

**http://www.netapp.com/us/library/white-papers/wp_3002.html**

**(At WPI: http://www.wpi.edu/Academics/CCC/Help/Unix/snapshots.html)**

# About the authors

- Dave Hitz, James Lau, and Michael Malcolm


- Founded NetApp in 1992
- NetApp is now a fortune 500 company worth $10 billion
- Malcolm left early, other two stuck around
- Current pics:

# Introduction

- In general, *appliance* is device designed to perform specific function
- Distributed systems trend has been to use appliances instead of general purpose computers. Examples:
  - *routers* from Cisco and Avici
  - network *terminals*
  - network *printers*
- For files, not just another computer with your files, but new type of network appliance

  → Network File System (NFS) file server

# Introduction: NFS Appliance

- NFS File Server Appliances have different requirements than those of general purpose file system
  - NFS access patterns are different than local file access patterns
  - Large client-side caches result in fewer reads than writes
- Network Appliance Corporation uses Write Anywhere File Layout (WAFL) file system

# Introduction: WAFL

- WAFL has 4 requirements
  - Fast NFS service
  - Support large file systems (10s of GB) that can grow (can add disks later)
  - Provide high performance writes and support Redundant Arrays of Inexpensive Disks (RAID)
  - Restart quickly, even after unclean shutdown
- NFS and RAID both strain write performance:
  - NFS server must respond after data is written
  - RAID must write parity bits also

# Outline

- Introduction (done)
- Snapshots : User Level (next)
- WAFL Implementation
- Snapshots: System Level
- Performance
- Conclusions

# Introduction to Snapshots

- *Snapshots* are copy of file system at given point in time
- WAFL creates and deletes snapshots automatically at preset times
  - Up to 255 snapshots stored at once
- Uses *copy-on-write* to avoid duplicating blocks in the active file system
- Snapshot uses:
  - Users can recover accidentally deleted files
  - Sys admins can create backups from running system
  - System can restart quickly after unclean shutdown
    - Roll back to previous snapshot

# User Access to Snapshots

- Example, suppose accidentally removed file named "`todo`":

```
CCCWORK3% ls -lut .snapshot/*/todo
-rw-rw---- 1 claypool claypool 4319 Oct 24 18:42
.snapshot/2011_10_26_18.15.29/todo
-rw-rw---- 1 claypool claypool 4319 Oct 24 18:42
.snapshot/2011_10_26_19.27.40/todo
-rw-rw---- 1 claypool claypool 4319 Oct 24 18:42
.snapshot/2011_10_26_19.37.10/todo
```

- Can then recover most recent version:

```
CCCWORK3% cp .snapshot/2011_10_26_19.37.10/todo todo
```

- Note, snapshot directories (`.snapshot`) are hidden in that they don't show up with `ls` (even `ls -a`) unless specifically requested

# Snapshot Administration

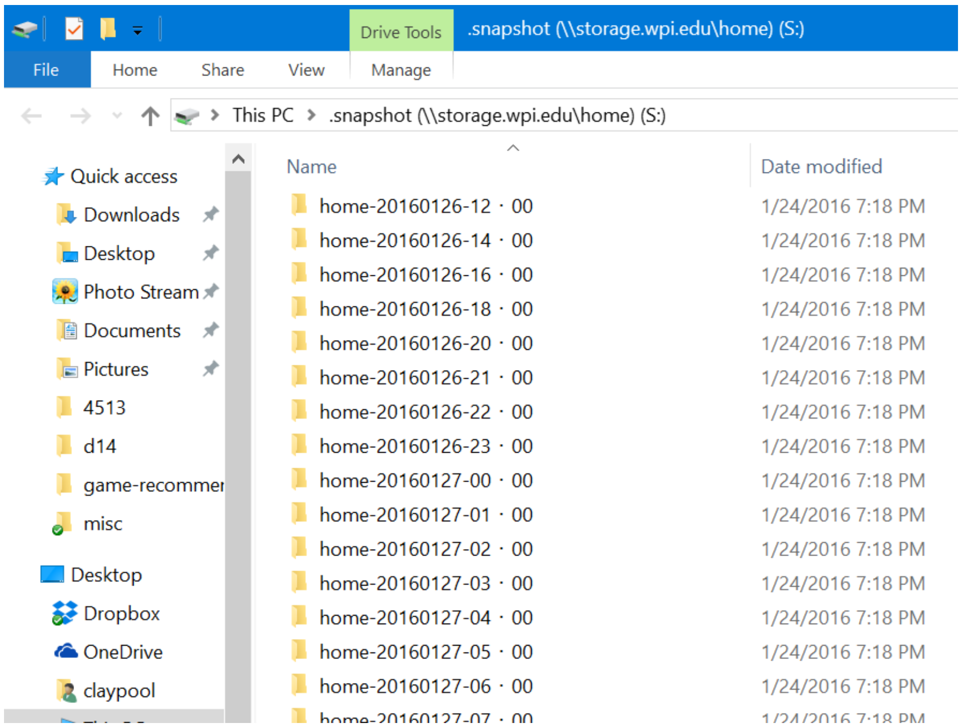
- WAFL server allows sys admins to create and delete snapshots, but usually automatic
- At WPI, snapshots of `/home`. Says:
  - 3am, 6am, 9am, noon, 3pm, 6pm, 9pm, midnight
  - Nightly snapshot at midnight every day
  - Weekly snapshot is made on Saturday at midnight every week

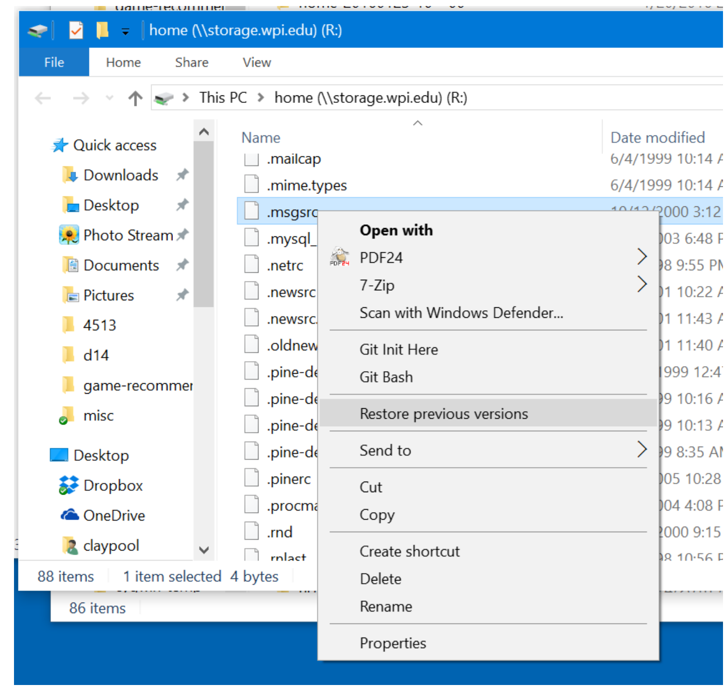
  → But looks like every 1 hour (fewer copies kept for older periods and 1 week ago max)

**claypool 168 CCCWORK3%** cd .snapshot
**claypool 169 CCCWORK3%** ls -1
home-20160121-00:00/
home-20160122-00:00/
home-20160122-22:00/
home-20160123-00:00/
home-20160123-02:00/
home-20160123-04:00/
home-20160123-06:00/
home-20160123-08:00/
home-20160123-10:00/
home-20160123-12:00/
…
home-20160127-16:00/
home-20160127-17:00/
home-20160127-18:00/
home-20160127-19:00/
home-20160127-20:00/
home-latest/

# Snapshots at WPI (Windows)

- Mount UNIX space (`\\storage.wpi.edu\home`), add `\.snapshot` to end




Note, files in `.snapshot` do not count against quota

- Can also right-click on file and choose "restore previous version"

# Outline

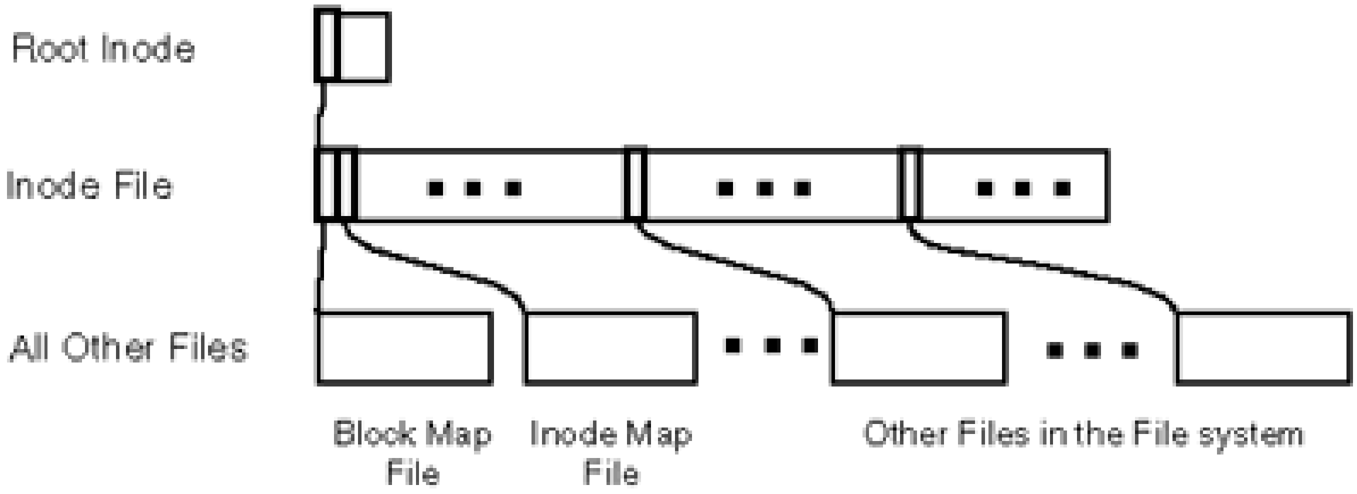
- Introduction (done)
- Snapshots : User Level (done)
- WAFL Implementation (next)
- Snapshots: System Level
- Performance
- Conclusions

# WAFL File Descriptors

- Inode based system with 4 KB blocks
- Inode has 16 pointers, which vary in type depending upon file size
  - For files smaller than 64 KB:
    - Each pointer points to data block
  - For files larger than 64 KB:
    - Each pointer points to indirect block
  - For really large files:
    - Each pointer points to doubly-indirect block
- For very small files (less than 64 bytes), data kept in inode itself, instead of using pointers to blocks

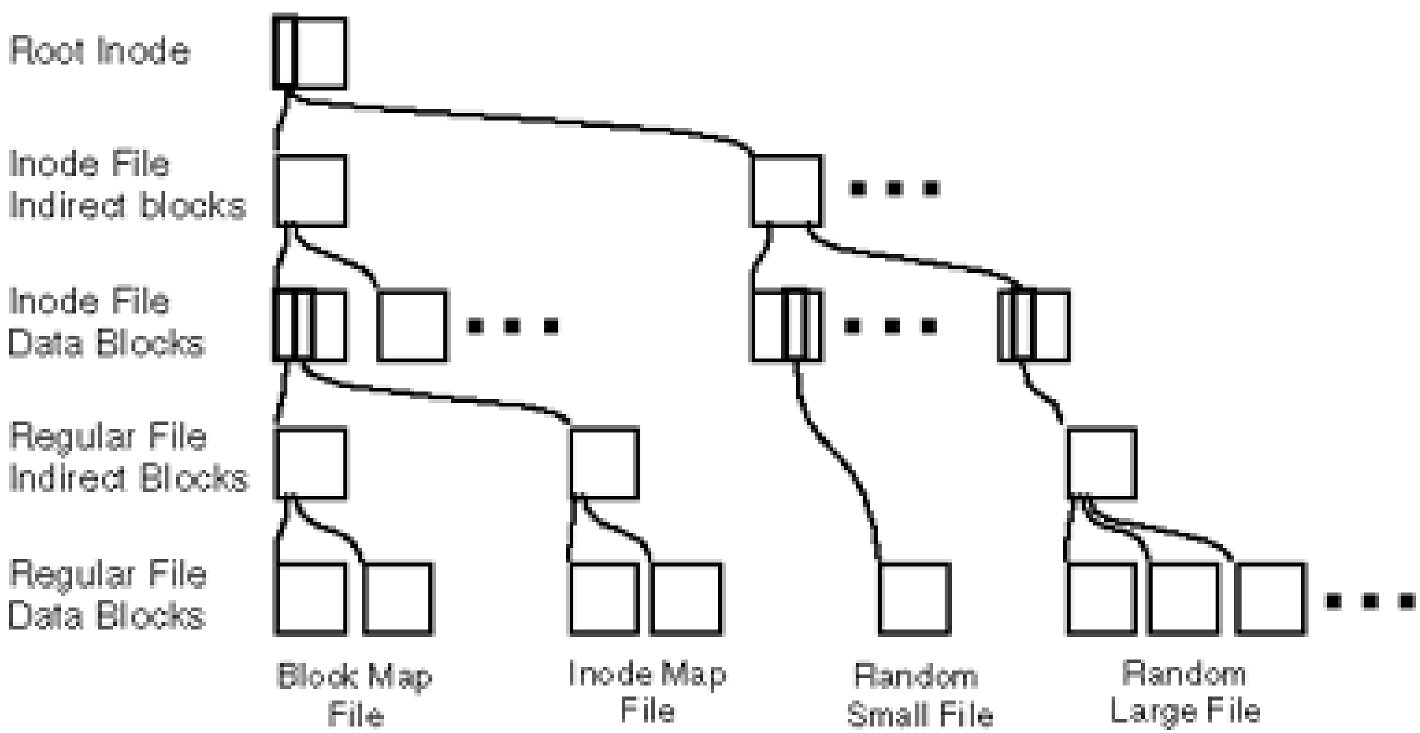
# WAFL Meta-Data

- Meta-data stored in files
  - Inode file – stores inodes
  - Block-map file – stores free blocks
  - Inode-map file – identifies free inodes



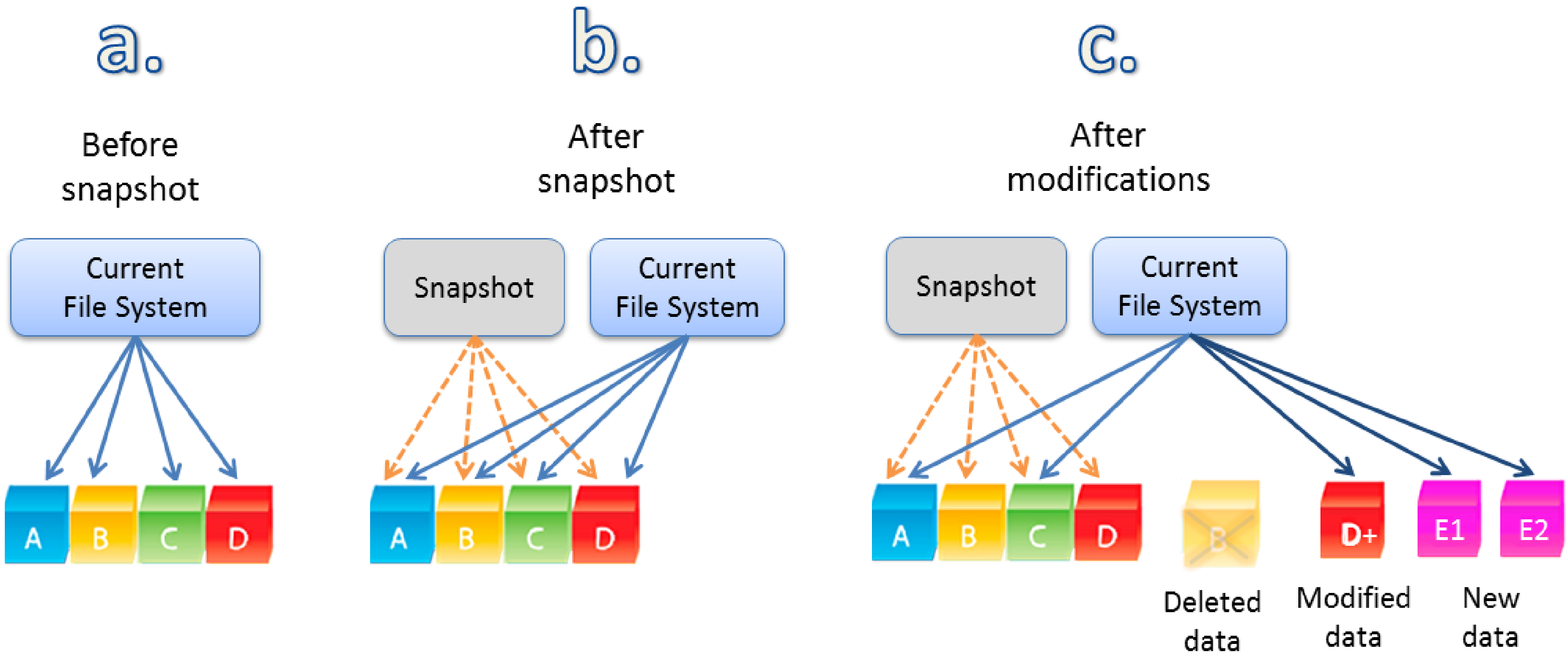
# Zoom of WAFL Meta-Data (Tree of Blocks)

- Root inode must be in fixed location
- Other blocks can be written anywhere



# Snapshots (1 of 2)

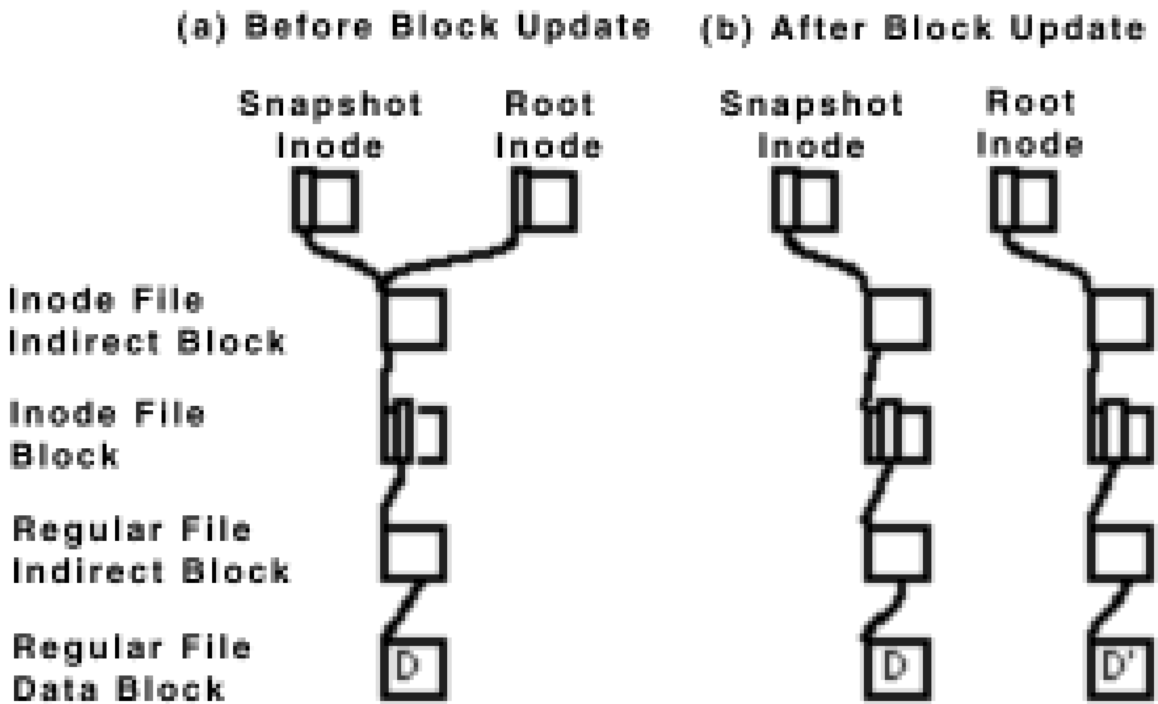
- Copy root inode only, copy on write for changed data blocks




- Over time, old snapshot references more and more data blocks that are not used
- Rate of file change determines how many snapshots can be stored on system

# Snapshots (2 of 2)

- When disk block modified, must modify meta-data (indirect pointers) as well




- Batch, to improve I/O performance

# Consistency Points (1 of 2)

- In order to avoid consistency checks after unclean shutdown, WAFL creates special snapshot called *consistency point* every few seconds
  - Not accessible via NFS
- Batched operations are written to disk each consistency point
  - Like journal
- In between consistency points, data only written to RAM

# Consistency Points (2 of 2)

- WAFL uses NVRAM (NV = Non-Volatile):
  - (NVRAM is DRAM with batteries to avoid losing during unexpected poweroff, some servers now just solid-state or hybrid)
  - NFS requests are logged to NVRAM
  - Upon unclean shutdown, re-apply NFS requests to last consistency point
  - Upon clean shutdown, create consistency point and turnoff NVRAM until needed (to save power/batteries)
- Note, typical FS uses NVRAM for metadata write cache instead of just logs
  - Uses more NVRAM space (WAFL logs are smaller)
    - Ex: “rename” needs 32 KB, WAFL needs 150 bytes
    - Ex: write 8 KB needs 3 blocks (data, inode, indirect pointer), WAFL needs 1 block (data) plus 120 bytes for log
  - Slower response time for typical FS than for WAFL (although WAFL may be a bit slower upon restart)

# Write Allocation

- Write times dominate NFS performance
  - Read caches at client are large
  - Up to 5*x* as many write operations as read operations at server
- WAFL batches write requests (e.g., at consistency points)
- WAFL allows "write anywhere", enabling inode next to data for better perf
  - Typical FS has inode information and free blocks at fixed location
- WAFL allows writes in any order since uses consistency points
  - Typical FS writes in fixed order to allow `fsck` to work if unclean shutdown

# Outline

- Introduction (done)
- Snapshots : User Level (done)
- WAFL Implementation (done)
- Snapshots: System Level (next)
- Performance
- Conclusions

# The Block-Map File

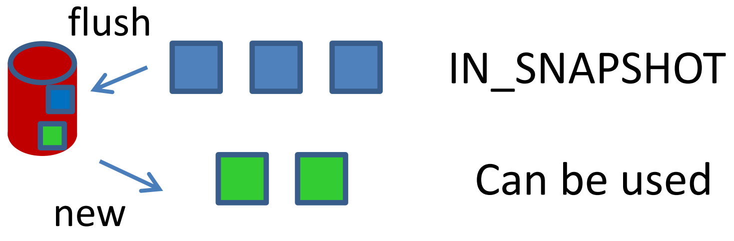
- Typical FS uses bit for each free block, 1 is allocated and 0 is free
  - Ineffective for WAFL since may be other snapshots that point to block
- WAFL uses 32 bits for each block
  - For each block, copy "active" bit over to snapshot bit

| Time | Block-Map Entry | Description |
|---|---|---|
| t1 | 00000000 | Block is unused |
| t2 | 00000001 | Block is allocated for active FS |
| t3 | 00000011 | Snapshot #1 is created |
| t4 | 00000111 | Snapshot #2 is created |
| t5 | 00000110 | Block is deleted from active FS |
| t6 | 00000110 | Snapshot #3 is created |
| t7 | 00000100 | Snapshot #1 is deleted |
| t8 | 00000000 | Snapshot #2 is deleted; block is unused |

bit 0: set for active file system
bit 1: set for Snapshot #1
bit 2: set for Snapshot #2
bit 3: set for Snapshot #3

# Creating Snapshots

- Could suspend NFS, create snapshot, resume NFS
  - But can take up to 1 second
- Challenge: avoid locking out NFS requests
- WAFL marks all dirty cache data as IN_SNAPSHOT. Then:
  - NFS requests can read system data, write data not IN_SNAPSHOT
  - Data not IN_SNAPSHOT not flushed to disk
- Must flush IN_SNAPSHOT data as quickly as possible



# Flushing IN_SNAPSHOT Data

- Flush inode data first
  - Keeps two caches for inode data, so can copy system cache to inode data file, unblocking most NFS requests
    - Quick, since requires no I/O since inode file flushed later
- Update block-map file
  - Copy active bit to snapshot bit
- Write all IN_SNAPSHOT data
  - Restart any blocked requests as soon as particular buffer flushed (don't wait for all to be flushed)
- Duplicate root inode and turn off IN_SNAPSHOT bit

- All done in less than 1 second, first step done in 100s of ms

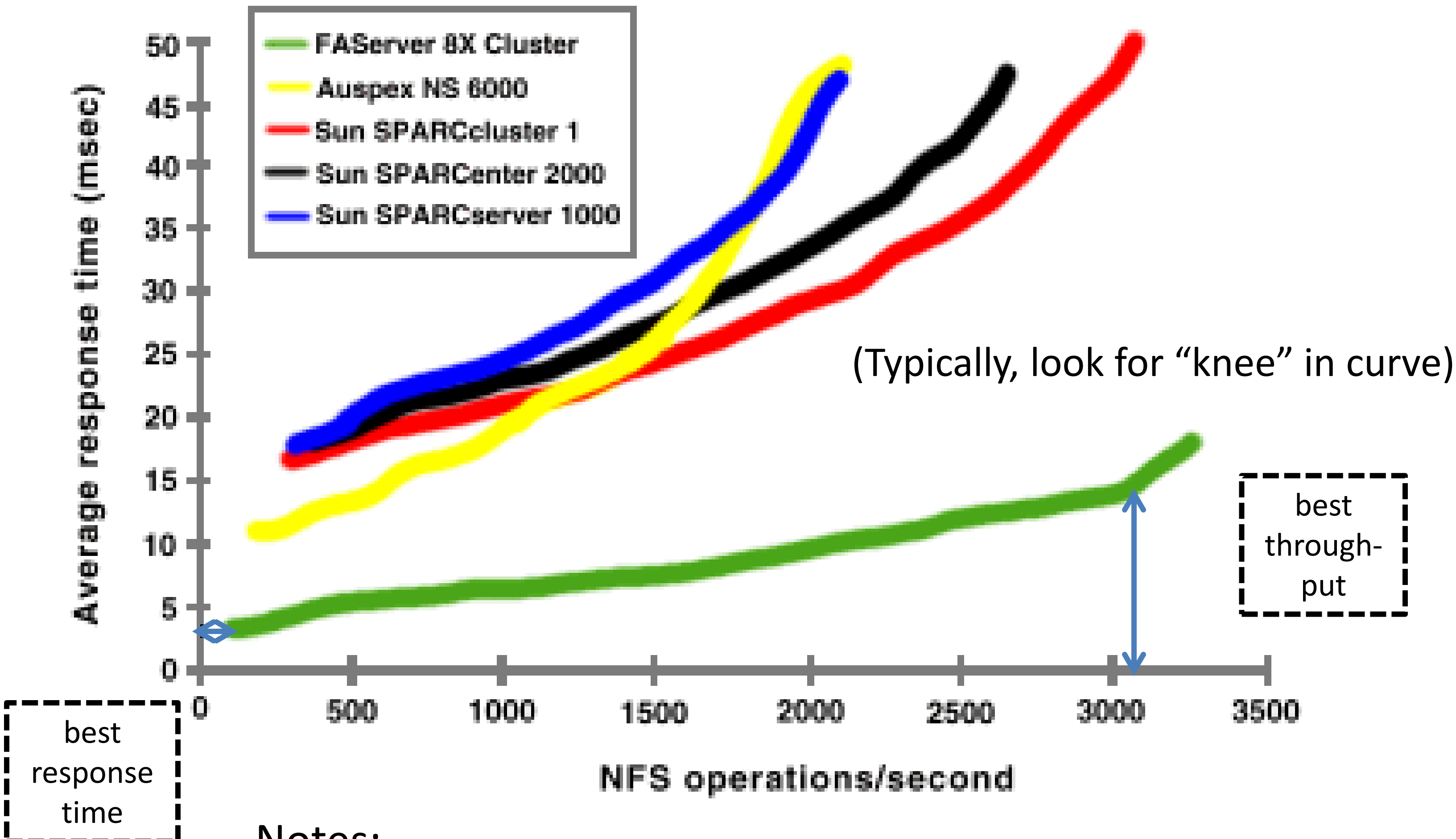
# Outline

- Introduction (done)
- Snapshots : User Level (done)
- WAFL Implementation (done)
- Snapshots: System Level (done)
- Performance (next)
- Conclusions

# Performance (1 of 2)

- Compare against other NFS systems
- How to measure NFS performance?
  - Best is SPEC NFS
    - LADDIS: Legato, Auspex, Digital, Data General, Interphase and Sun
- Measure response times versus throughput
  - Typically, servers quick at low throughput then response time increases as throughput requests increase
- (Me: System Specifications?!)

# Performance (2 of 2)



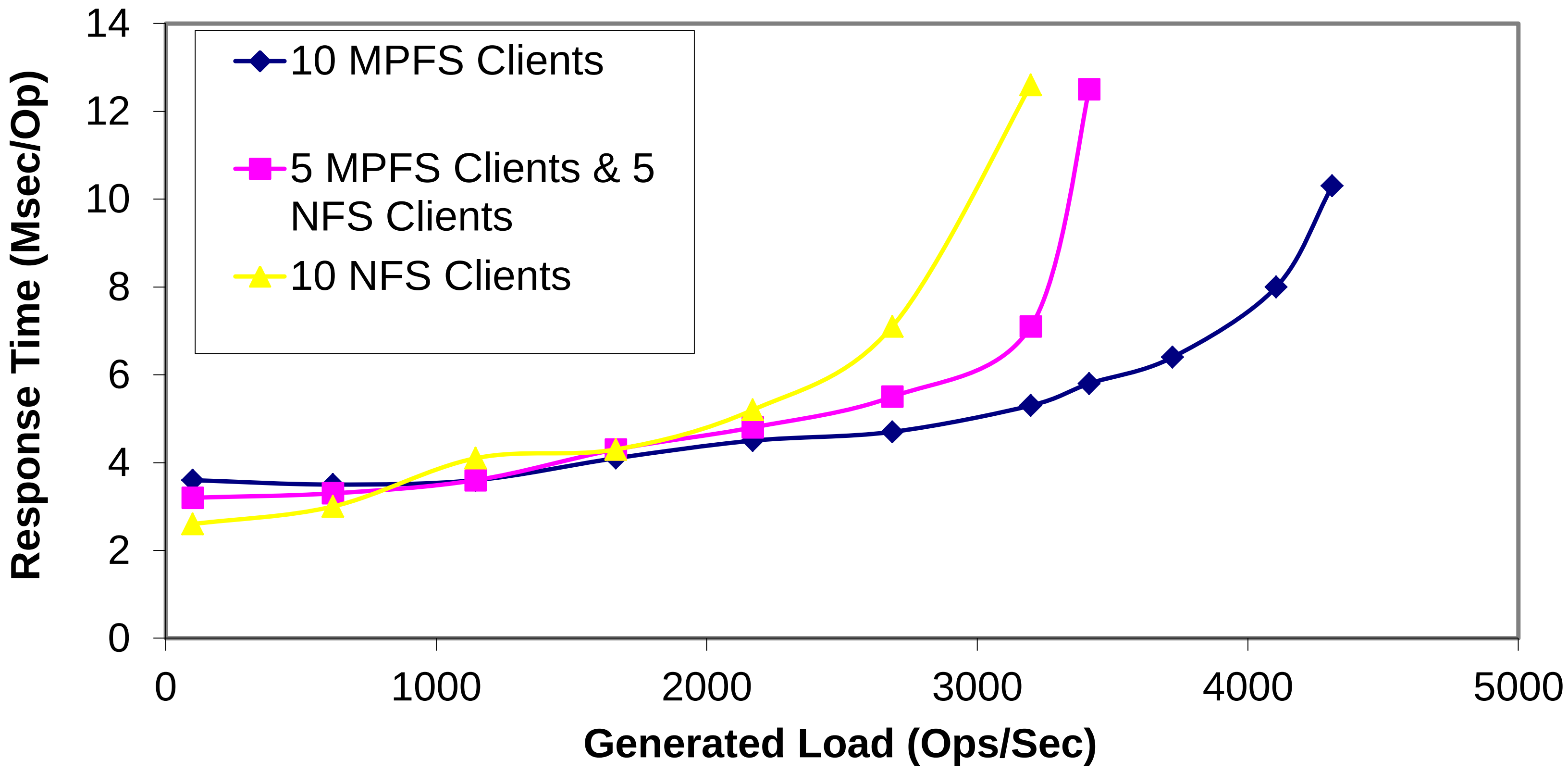

Notes:

+ FAS has only 8 file systems, and others have dozens

- FAS tuned to NFS, others are general purpose

# NFS vs. Newer File Systems

MPFS = multi-path file system
Used by EMC Celerra




- Remove NFS server as bottleneck
- Clients write directly to device

# Conclusion

- NetApp (with WAFL) works and is stable
  - Consistency points simple, reducing bugs in code
  - Easier to develop stable code for network appliance than for general system
    - Few NFS client implementations and limited set of operations so can test thoroughly
- WPI bought one ☺

# Later NetApp/WAFL capabilities

- What if we make a big file on a WAFL file system, then treat that file as a virtual block device, and we make a WAFL file system on that?
  - Now file systems can dynamically grow and shrink (because they're really files)
  - Can do some optimizations to reduce the overhead of going through two file system layers: inner file system can be "aware" that it's hosted on an outer file system
  - Result: **thin provisioning** – Allocate more storage than you've got
- Similarly, LUNs are just fixed-size files
  - Result: **SAN support**
- Multiple files can refer to same data blocks with copy-on-write semantics
  - Result: **writable clones**

# ZFS

- Copy-on-Write functions similar to WAFL
  - Similar enough that NetApp sued Sun over it...
- Integrates Volume Manager & File System
  - Software RAID without the write hole
- Integrates File System & Buffer Management
  - Advanced prefetching: strided patterns etc.
  - Use Adaptive Replacement Cache (ARC) instead of LRU
- File System reliability
  - Check summing of all data and metadata
  - Redudant Metadata

# Conclusion

- File system design is a major contributor to overall performance

- File system can provide major differentiating features
  - Do things that you didn't know you wanted to do (snapshots, clones, etc.)