File systems

CS 241

May 2, 2014

University of Illinois

Announcements

Finals approaching, know your times and conflicts

- Ours: Friday May 16, 8-11 am
- Inform us by Wed May 7 if you have to take a conflict exam (with justification)
- Review information TBA

Honors section demos

- Email us your group's availability by Sunday May 4
- Follow the specific instructions on Piazza
  - https://piazza.com/class/ho7vaxphwq9283?cid=938

MP Grading Round 2

- Formula: score = max{ old, new/2 }

Filesystems

A filesystem provides a high-level application access to storage media

- Masks the details of low-level sector-based I/O operations
- Provides structured access to data (files and directories)
- Caches recently-accessed data in memory

Most common general-purpose file systems

- Organized as a tree of directories and files
- Byte-oriented: may read and write files a byte at a time
- Alternate models exist
  - (key, value) storage instead of hierarchy
  - record-oriented files (like a data structure in a file) instead of a flat stream of bytes

Versioning filesystems

- Keep track of older versions of files
- e.g., VMS filesystem: Could refer to specific file versions: `foo.txt;1`, `foo.txt;2`

Filesystem Operations

Filesystems provide a standard interface to files and directories:

- Create a file or directory
- Delete a file or directory
- Open a file or directory – allows subsequent access
- Read, write, append to file contents
- Add or remove directory entries
- Close a file or directory – terminates access

What other features do filesystems provide?

- Accounting and quotas – prevent your classmates from hogging the disks
- Backup – some filesystems have a "$HOME/.backup" containing automatic snapshots
- Indexing and search capabilities
- File versioning
- Encryption
- Automatic compression of infrequently-used files
- Caching in memory

Should this functionality be part of the filesystem or built on top?

- Classic OS community debate: Where is the best place to put functionality?

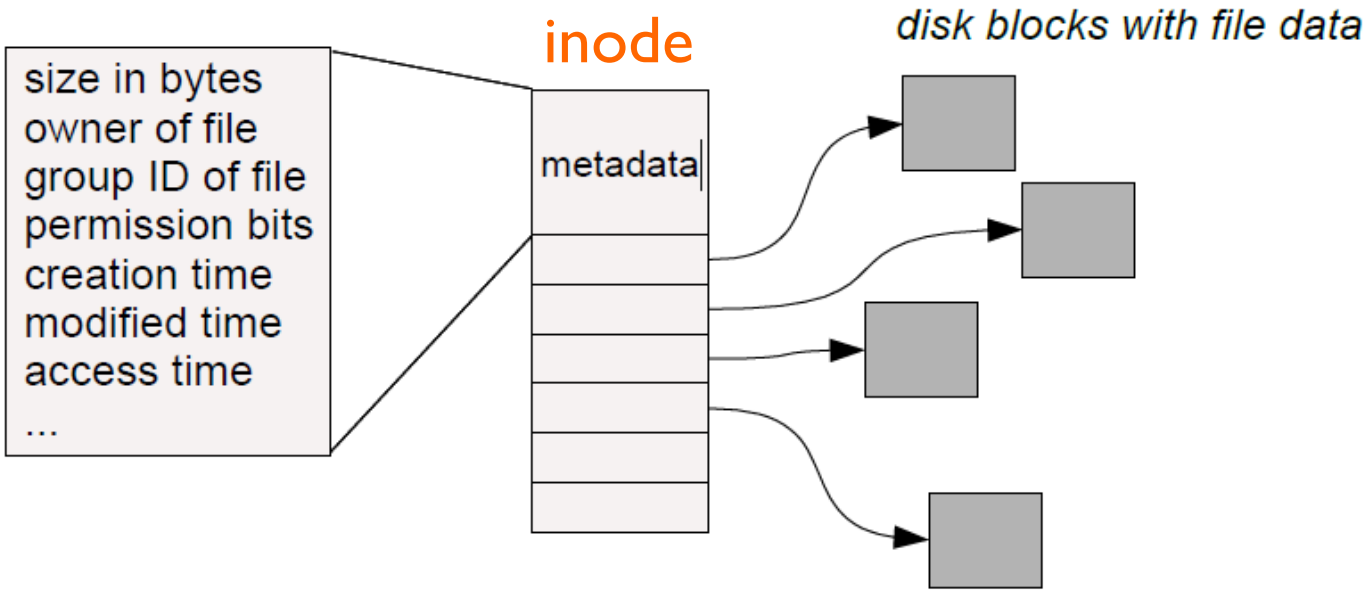
Basic Filesystem Structures

Every file and directory is represented by an inode

- Stands for "index node"

Contains two kinds of information:

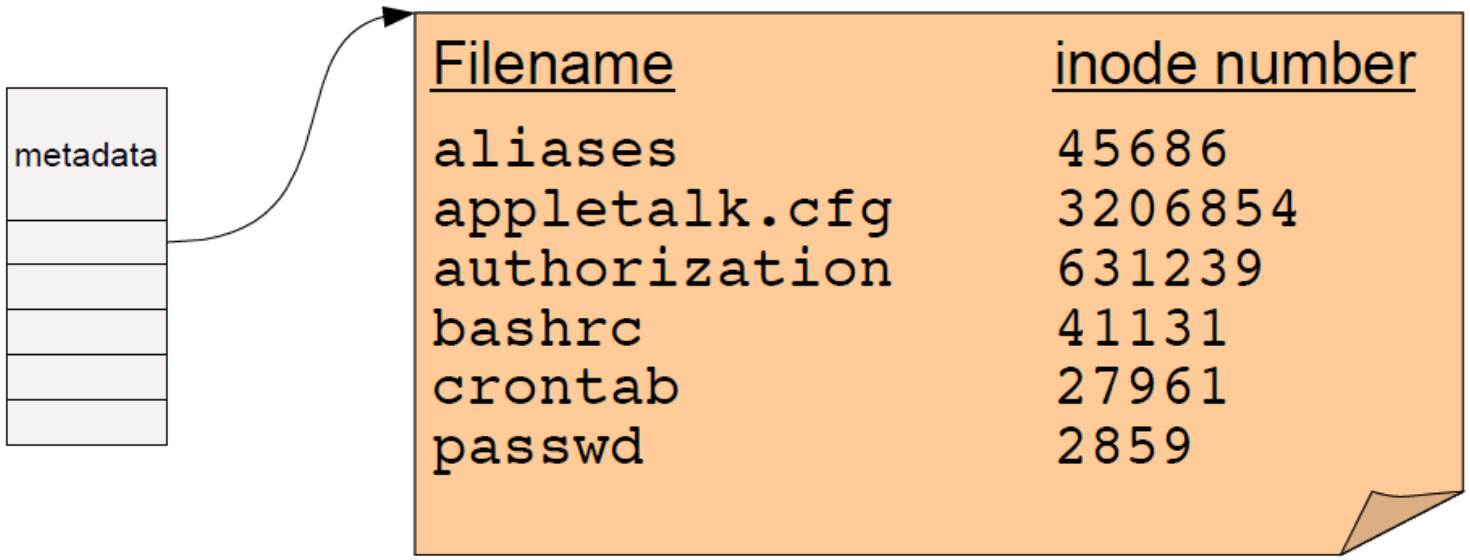
- 1) Metadata describing the file's owner, access rights, etc.
- 2) Location of the file's blocks on disk



Directories

A directory is a special kind of file that contains a list of (filename, inode number) pairs

- These are the contents of the directory "file data" itself – NOT the directory's inode!
- Filenames (in UNIX) are not stored in the inode at all!



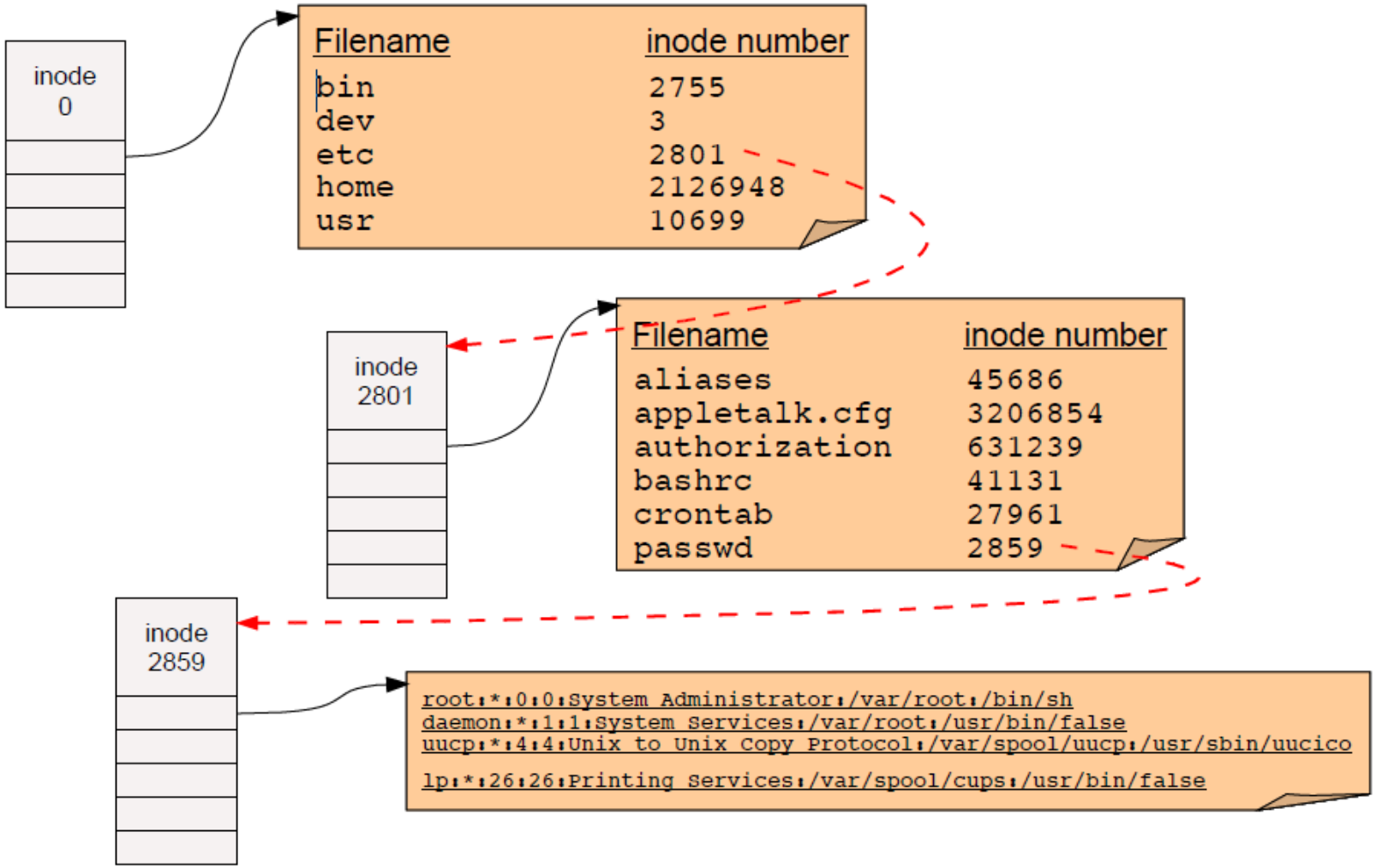
Finding a file

Two open questions:

- How do we find the root directory (“ / ” on UNIX systems)?
- How do we get from an inode number to the location of the inode on disk?

Resolving "/etc/password"

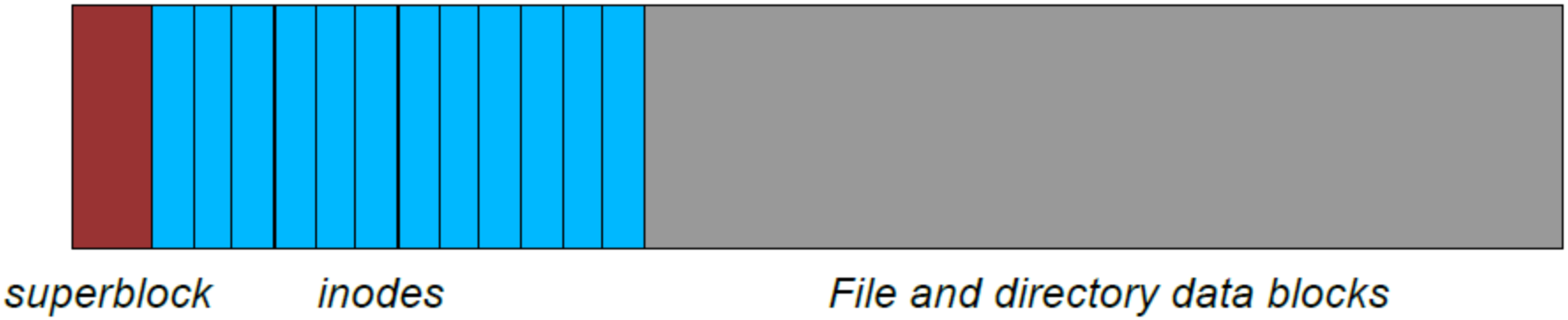
Start at root directory and walk down chain of inodes



Locating inodes on disk

All right, so directories tell us the inode number of a file.

- How the heck do we find the inode itself on disk?



- inode number is just the "index" of the inode
- Easy to compute the block address of a given inode:
  - block_addr(inode_num) = block_offset_of_first_inode + (inode_num * inode_size)
- This implies that a filesystem has a fixed number of potential inodes
  - This number is generally set when the filesystem is created
- The superblock stores important metadata on filesystem layout, list of free blocks, etc.

Stupid directory tricks

Directories map filenames to inode numbers. What does this imply?

We can create multiple pointers to the same inode in different directories

- Or even the same directory with different filenames

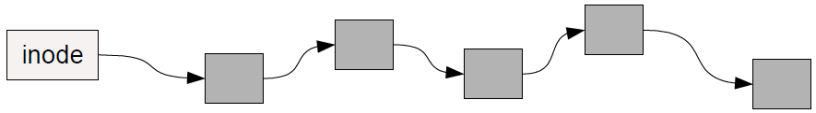
In UNIX this is called a "hard link" and can be done using "ln"

  - bash$ ls -i /home/foo
  - 287663 /home/foo (This is the inode number of "foo")
  - bash$ ln /home/foo /tmp/foo
  - bash$ ls -i /home/foo /tmp/foo
  - 287663 /home/foo
  - 287663 /tmp/foo
- "/home/foo" and "/tmp/foo" now refer to the same file on disk
- Not a copy! You will always see identical data no matter which filename you use to read or write the file.
- Not the same as a "symbolic link"! That links one filename to another.

How should we organize blocks on a disk?

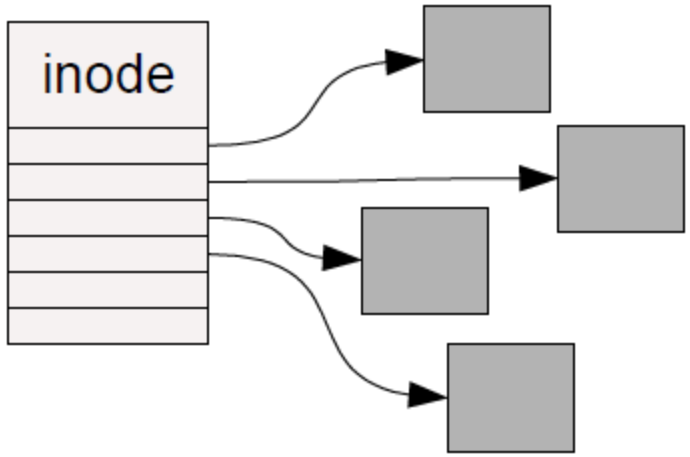
Very simple policy: A file consists of linked blocks

- inode points to the first block of the file
- Each block points to the next block in the file (just a linked list on disk)
  - What are the advantages and disadvantages??



Indexed files

- inode contains a list of block numbers containing the file
- Array is allocated when the file is created
  - What are the advantages and disadvantages??



Multilevel indexed files

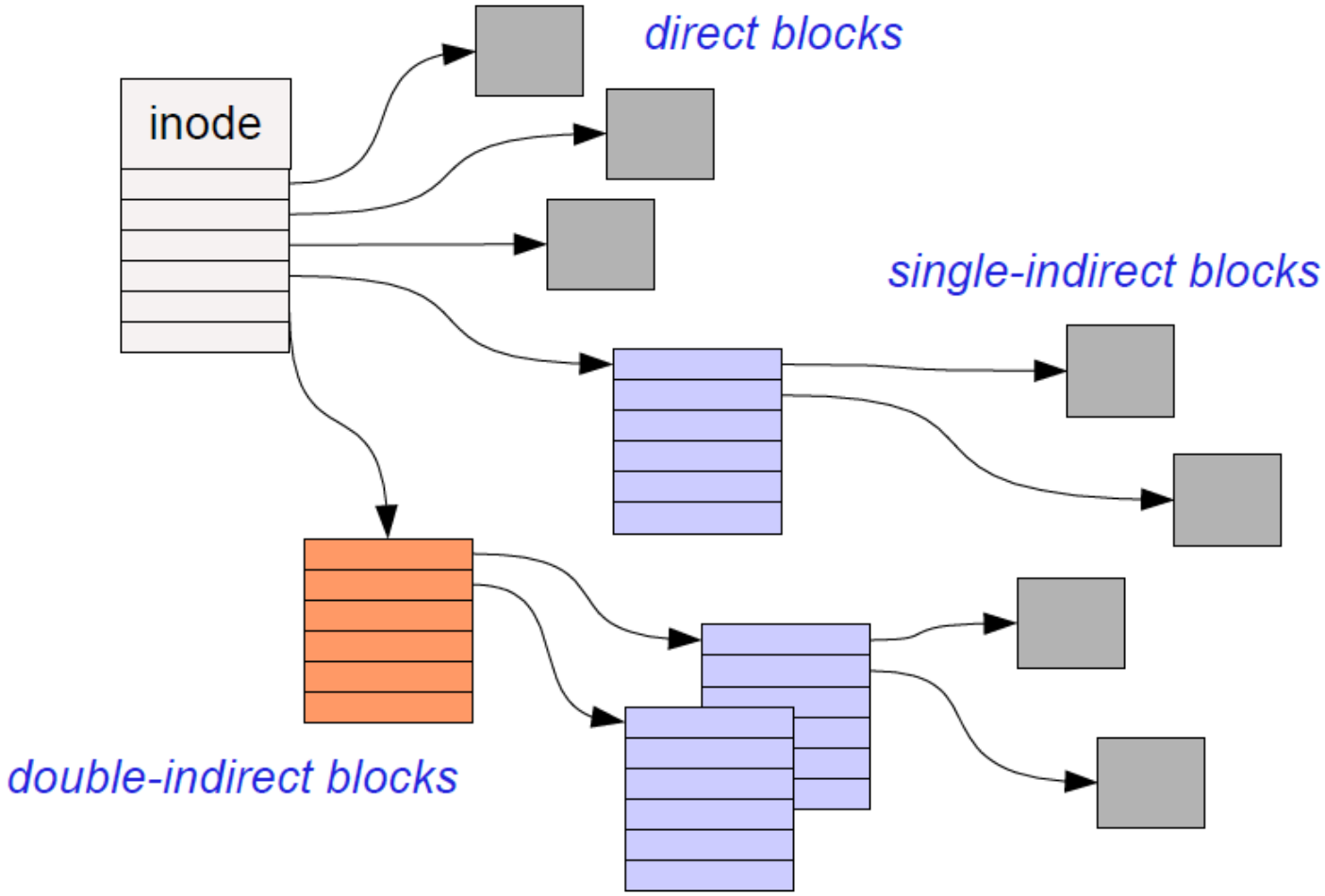
inode contains a list of 10-15 direct block pointers

- First few blocks of file can be referred to by the inode itself

inode also contains a pointer to a single indirect, double indirect, and triple indirect blocks

- Allows file to grow to be incredibly large!!!

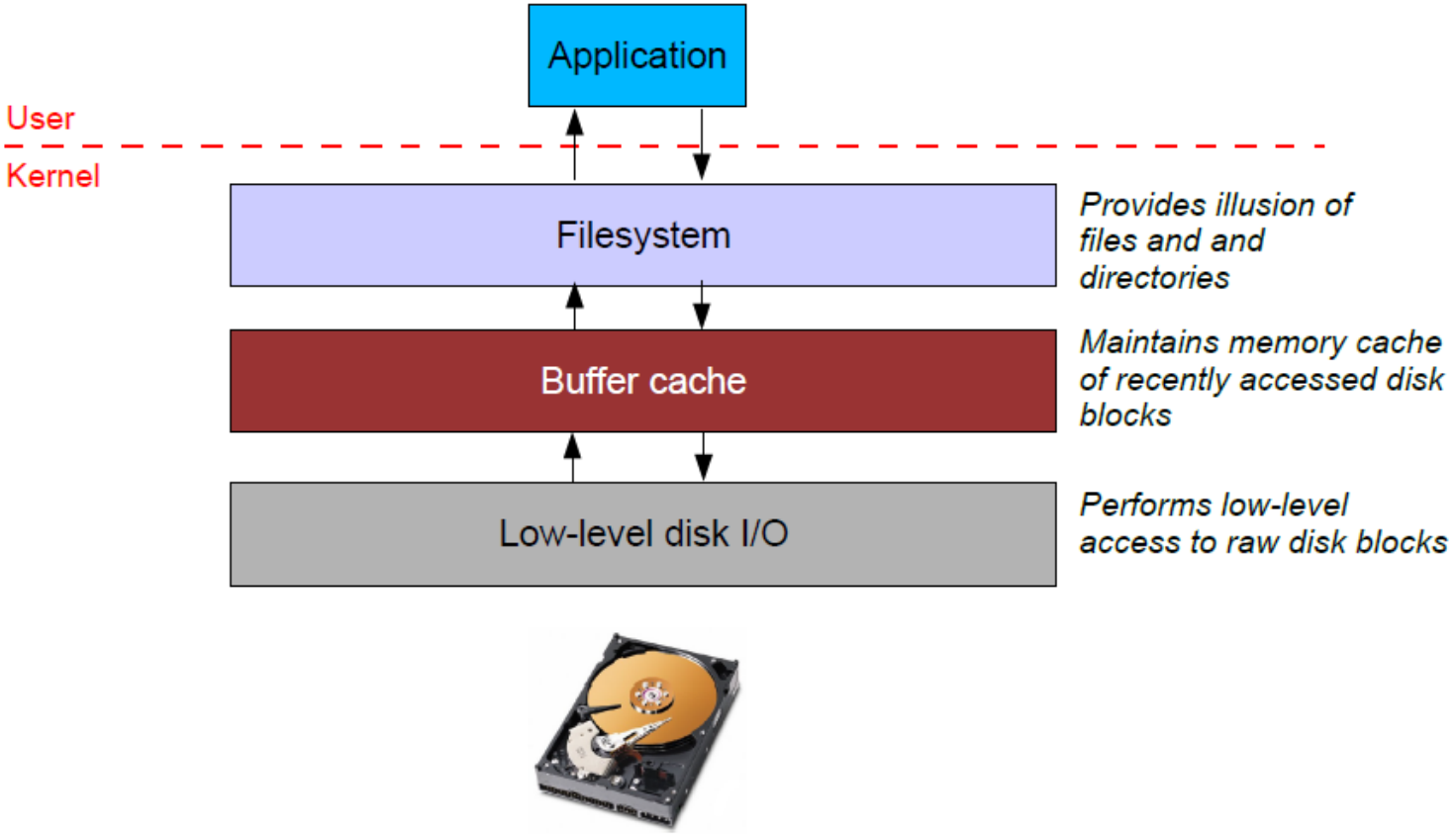
Multilevel indexed files



File system caching

Most filesystems cache significant amounts of disk in memory

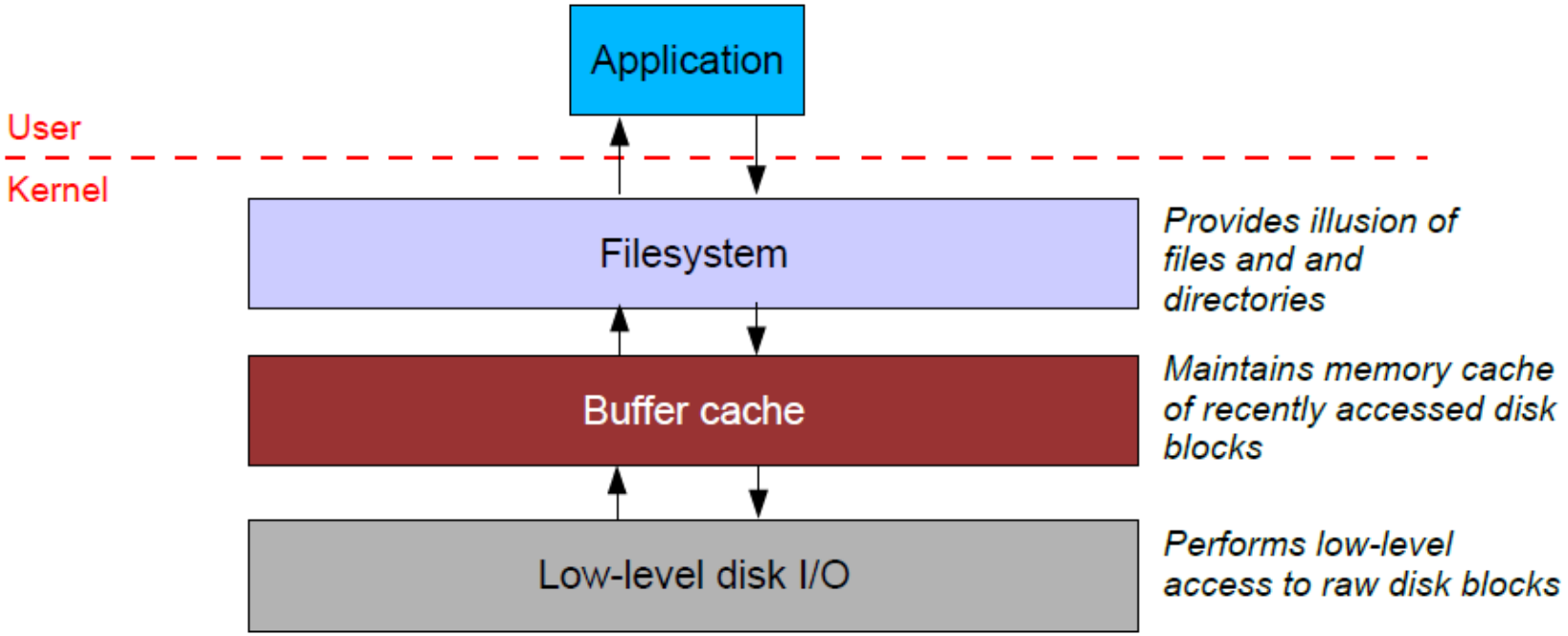
- e.g., Linux tries to use all “free” physical memory as a giant cache
- Avoids huge overhead for going to disk for every I/O



Caching issues (2)

Reliability issues

- What happens when you write to the cache but the system crashes?
- What if you update some of the blocks on disk but not others?
  - Example: Update the inode on disk but not the data blocks?
- Write-through cache: All writes immediately sent to disk
- Write-back cache: Cache writes stored in memory until evicted (then written to disk)
  - Tradeoffs?



Caching issues (2)

"Syncing" a filesystem writes back any dirty cache blocks to disk

- UNIX "`sync`" command achieves this.
- Can also use `fsync()` system call to sync any blocks for a given file.
  - Warning – not all UNIX systems guarantee that after sync returns that the data has really been written to the disk!
  - This is also complicated by memory caching on the disk itself.

Crash recovery

- If system crashes before sync occurs, a tool like `fsck` checks the filesystem for errors
- Example: an inode pointing to a block that is marked as free in the free block list
- Another example: An inode with no directory entry pointing to it
  - These usually get linked into a "lost+found" directory
  - inode does not contain the filename so need the sysadmin to look at the file dataand guess where it might belong!

Caching and fsync() example

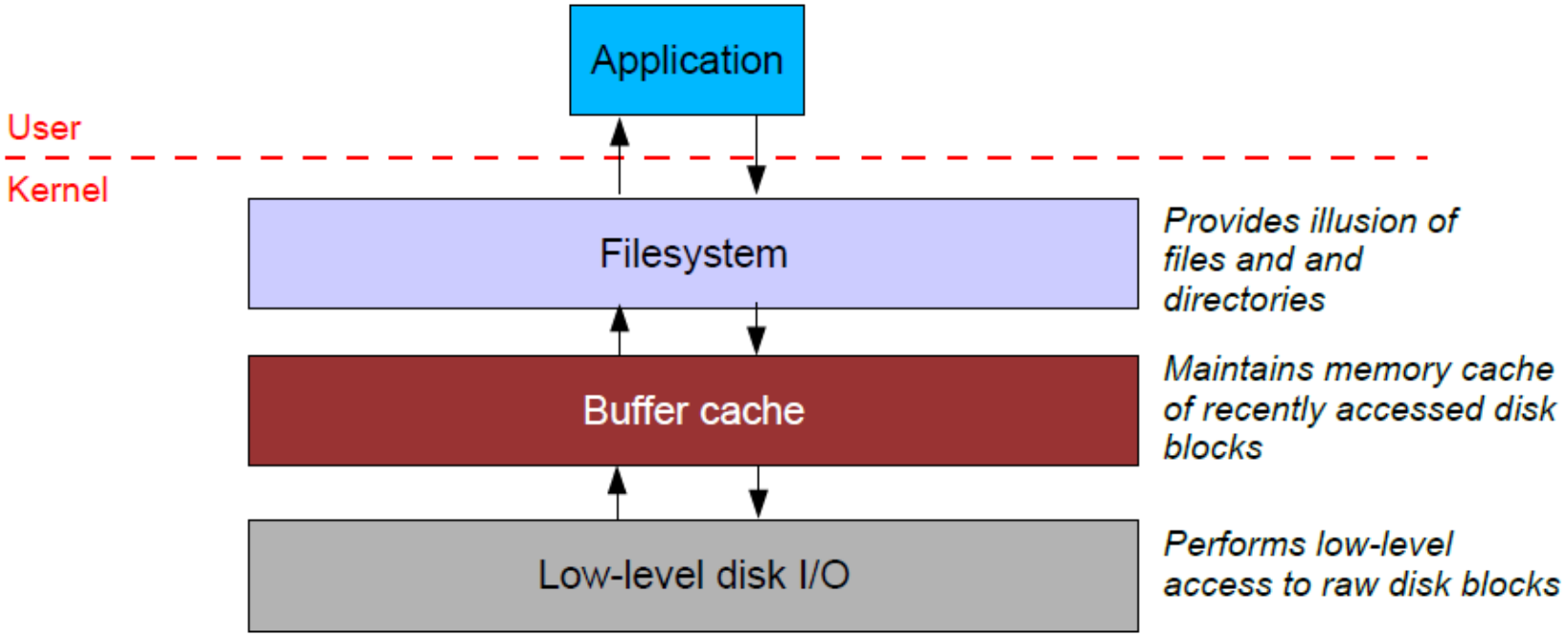
Running the copy example from last time,

- How fast is it the first time, vs. the second time you copy the same file?
- What happens if we fsync() after each iteration?

Caching issues (3)

Read ahead

- Seek time dominates overhead of disk I/O
- So, would ideally like to read multiple blocks into memory when you have a cache miss
  - Amortize the cost of the seek for multiple reads
- Useful if file data is laid out in contiguous blocks on disk
  - Especially if the application is performing sequential access to the file



Making filesystems resilient: RAID

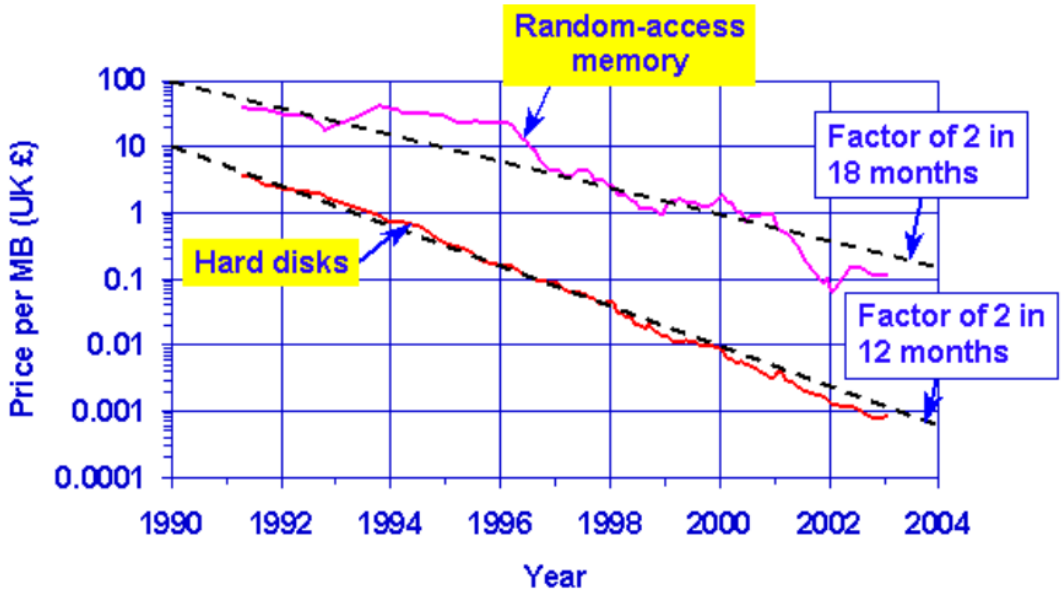
RAID Motivation

Speed of disks not matching other components

- Moore's law: CPU speed doubles every 18 months
- SRAM speeds increasing by 40-100% a year
- In contrast, disk seek time only improving 7% a year
  - Although greater density leads to improved transfer times once seek is done

Emergence of PCs starting to drive down costs of disks

- (This is 1988 after all)
- PC-class disks were smaller, cheaper, and only marginally slower



RAID Motivation

Basic idea: Build I/O systems as arrays of cheap disks

- Allow data to be striped across multiple disks
- Means you can read/write multiple disks in parallel – greatly improve performance

Problem: disks are extremely unreliable

Mean Time to Failure (MTTF)

- MTTF (disk array) = MTTF (single disk) / # disks
- Adding more disks means that failures happen more frequently..
- An array of 100 disks with an MTTF of 30,000 hours = just under 2 weeks for the array's MTTF!

Increasing reliability

Idea: Replicate data across multiple disks

- When a disk fails, lost information can be regenerated from the redundant data

Simplest form: Mirroring (also called “RAID 1”)

- All data is mirrored across two disks

Advantages:

- Reads are faster, since both disks can be read in parallel
- Higher reliability (of course)

Disadvantages:

- Writes are slightly slower, since OS must wait for both disks to do write
- Doubles the cost of the storage system!

RAID 3

Rather than mirroring, use parity codes

- Given N bits $\{b_1, b_2, ..., b_N\}$, the parity bit P is the bit {0,1} that yields an even number of "1" bits in the set $\{b_1, b_2, ..., b_N, P\}$
- Idea: If any bit in $\{b_1, b_2, ..., b_N\}$ is lost, can use the remaining bits (plus P) to recover it.

Where to store the parity codes?

- Add an extra "check disk" that stores parity bits for the data stored on the rest of the N disks

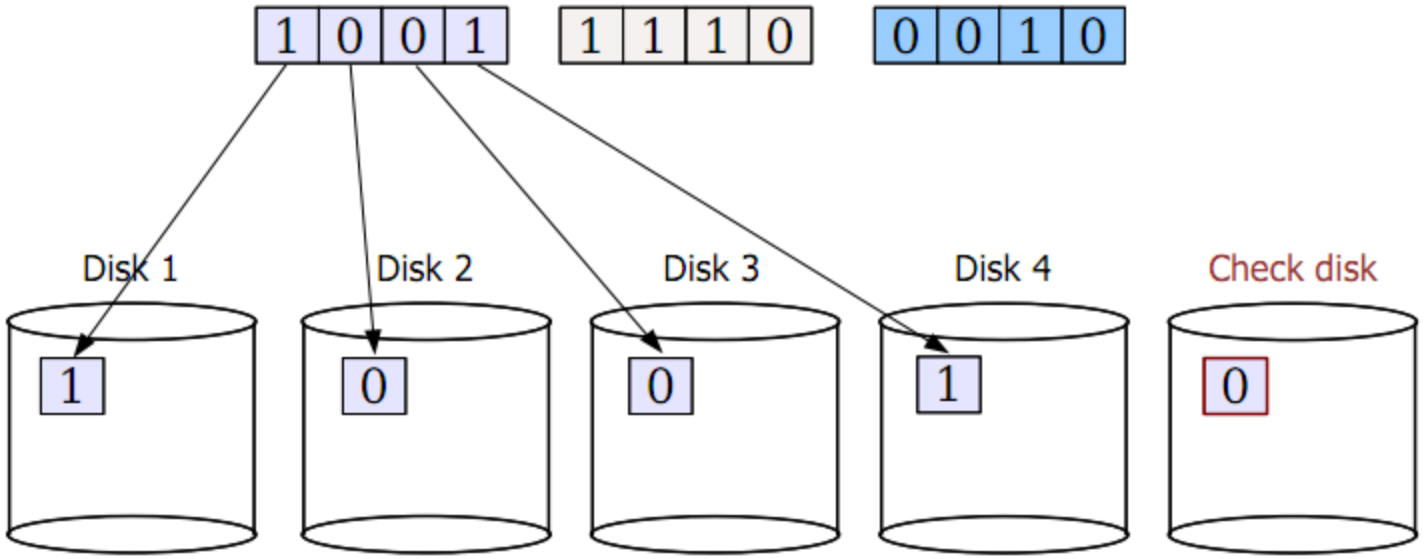
Advantages:

- If a single disk fails, can easily recompute the lost data from the parity code
- Can use one parity disk for several data disks (reduces cost)

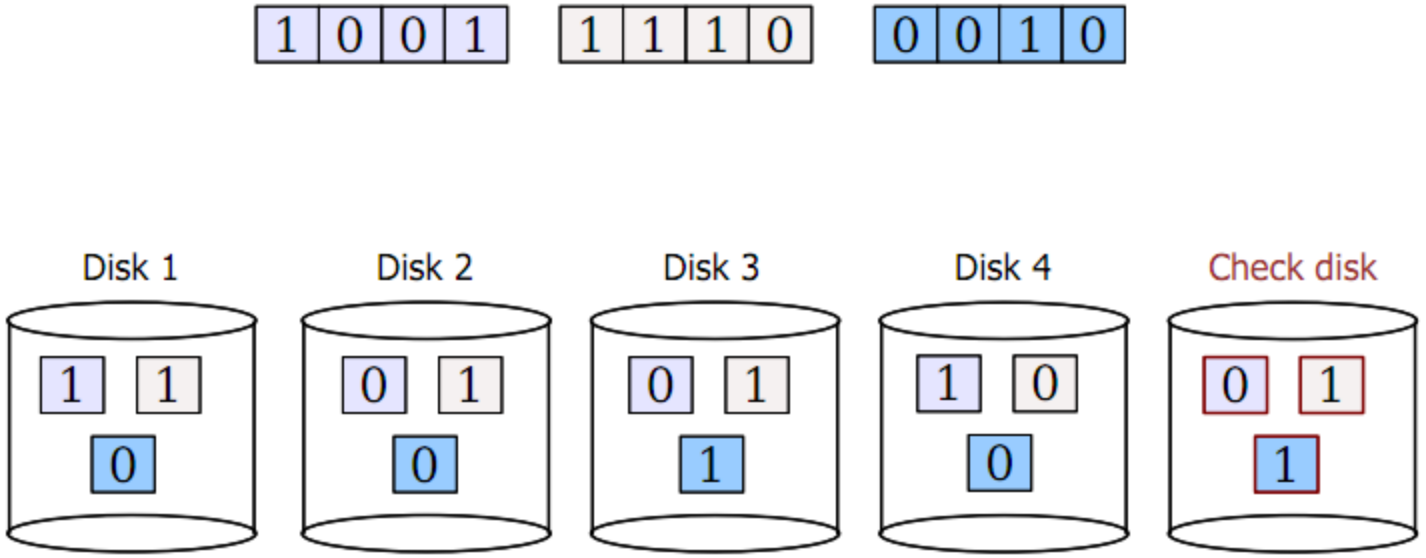
Disadvantages:

- Each write to a block must update the corresponding parity block as well

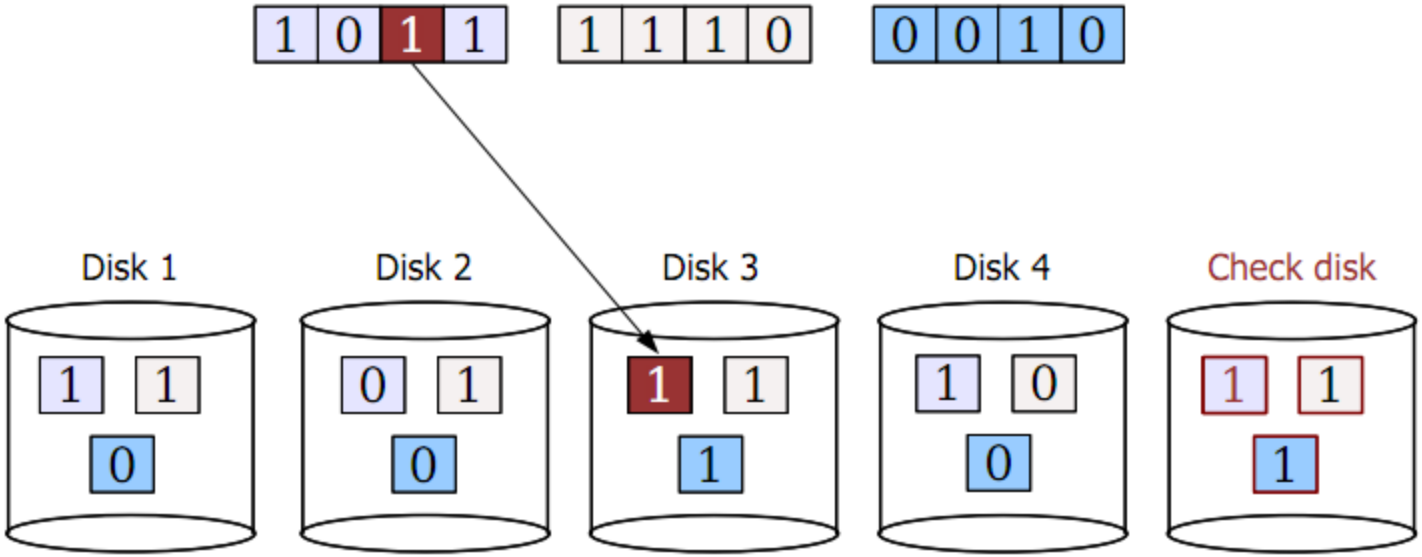
RAID 3 example



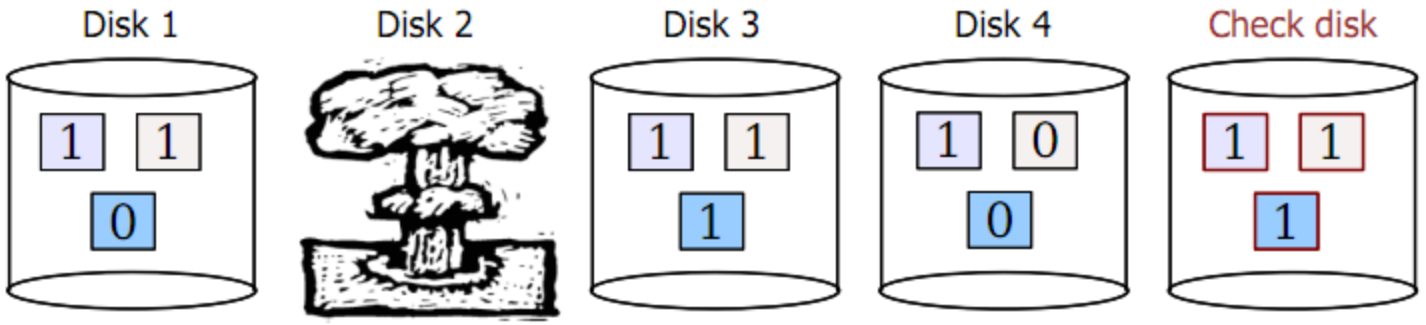
RAID 3 example



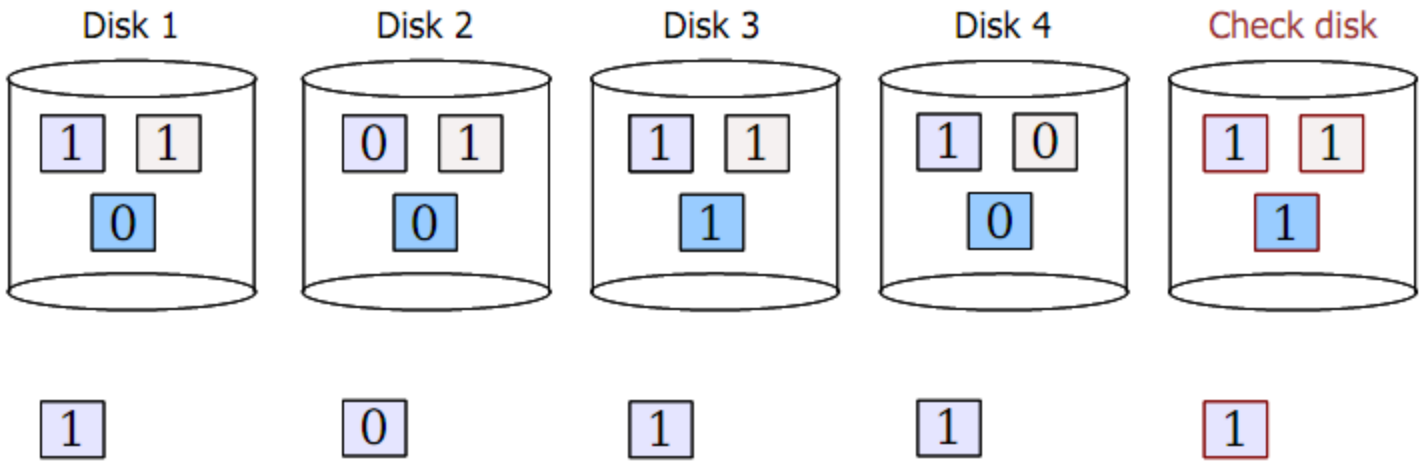
RAID 3 example



RAID 3 example



RAID 3 example



1. Read back data from other disks
2. Recalculate lost data from parity code
3. Rebuild data on lost disk

RAID 3 issues

Terminology

- MTTF = mean time to failure
- MTTR = mean time to repair

What is the MTTF of RAID?

- Both RAID 1 and RAID 3 tolerate the failure of a single disk
- As long as a second disk does not die while we are repairing the first failure, we are in good shape!

So, what is the probability of a second disk failure?

P(2nd failure) ≈ MTTR / (MTTF of one disk / # disks -1)

  - Assumes independent, exponential failure rates; see Patterson RAID paper for derivation
- 10 disks, MTTF (disk) = 1000 days, MTTR = 1 day
  - P(2nd failure) ≈ 1 day / (1000 / 9) = 0.009

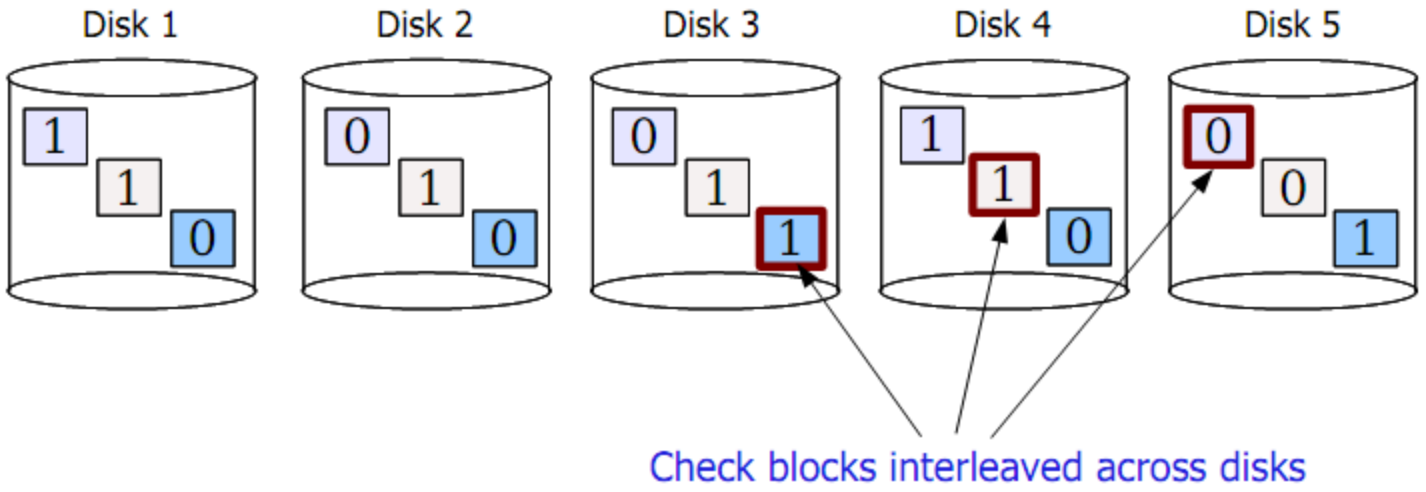
What is the performance of RAID 3?

- Check disk must be updated each time there is a write
- Problem: The check disk is then a performance bottleneck
  - Only a single read/write can be done at once on the whole system!

RAID 5

Another approach: Interleaved check blocks ("RAID 5")

- Rotate the assignment of data blocks and check blocks across disks
- Avoids the bottleneck of a single disk for storing check data
- Allows multiple reads/writes to occur in parallel (since different disks affected)



Reliable distributed storage

Today, giant data stores distributed across 100s of thousands of disks across the world

- e.g., your mail on gmail

"You know you have a large storage system when you get paged at 1 AM because you only have a few petabytes of storage left."

- – a "note from the trenches" at Google

Reliable distributed storage

Issues

- Failure is the common case
  - Google reports 2-10% of disks fail per year
  - Now multiply that by 60,000+ disks in a single warehouse...
- Must survive failure of not just a disk, but a rack of servers or a whole data center

Solutions

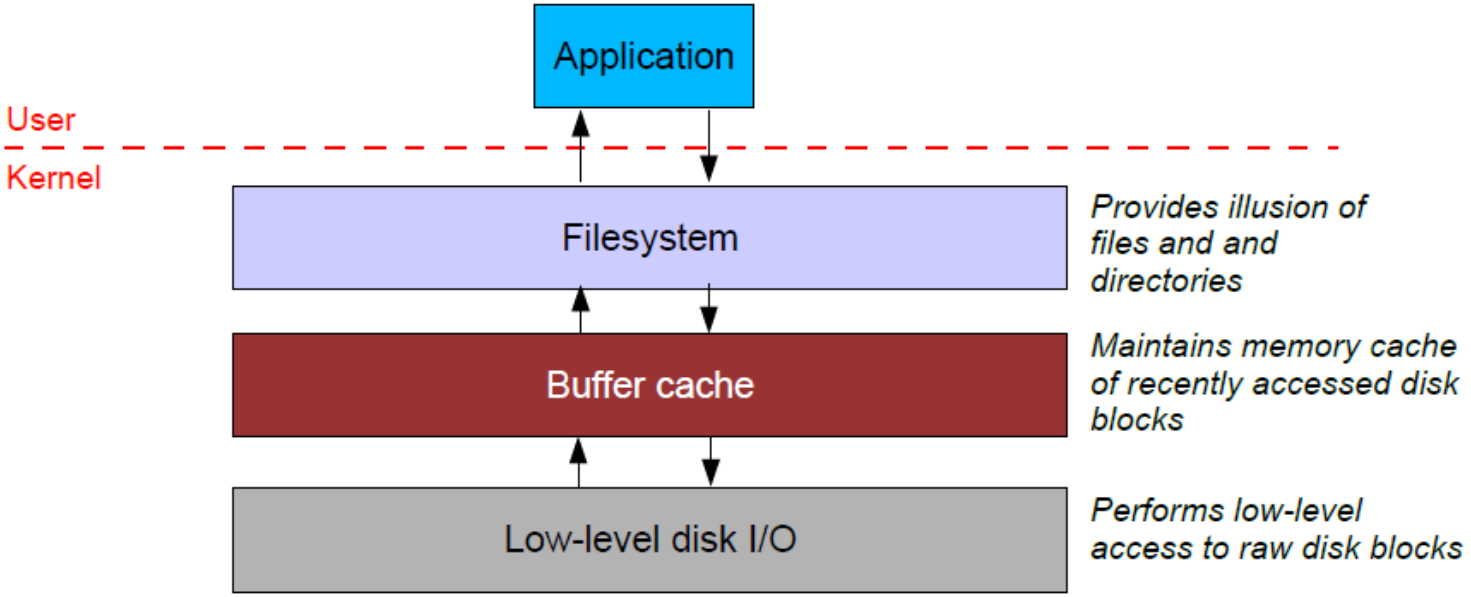
- Simple redundancy (2 or 3 copies of each file)
  - e.g., Google GFS (2001)
- More efficient redundancy (analogous to RAID 3++)
  - e.g., Google Colossus filesystem (~2010): customizable replication including Reed-Solomon codes with 1.5x redundancy

More interesting tidbits: http://goo.gl/LwFly

Caching issues

Where should the cache go?

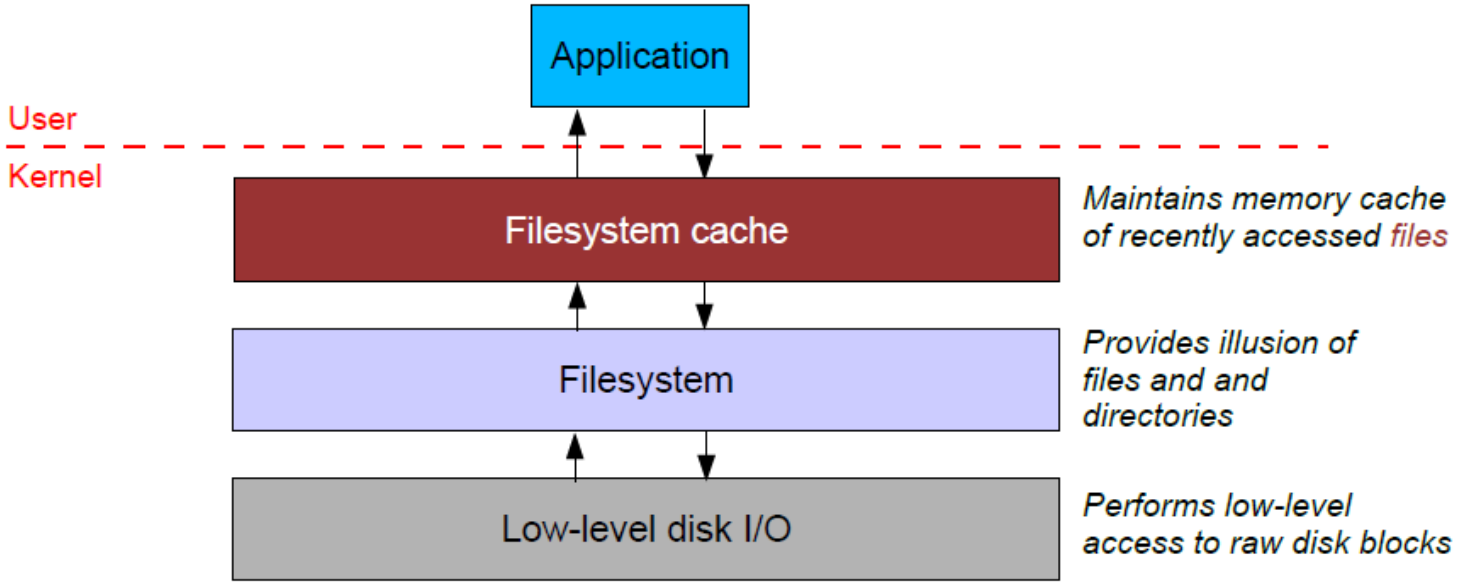
- Below the filesystem layer: Cache individual disk blocks
- Above the filesystem layer: Cache entire files and directories
- Which is better??



Caching issues

Where should the cache go?

- Below the filesystem layer: Cache individual disk blocks
- Above the filesystem layer: Cache entire files and directories
- Which is better??



34