CPSC313: Computer Hardware and Operating Systems

Unit 4: File Systems
Representing files on disk

Admin

- Final examination
  - Reserve your time of PrairieTest if you haven't already done it.
- Quiz 3 retakes
  - From yesterday until Saturday
- Tutorial 8 is this week
- Lab 8 has been released and is due Sunday November 17th.
- Code for today is in the course code repository: 4.3-file-indexing

More admin

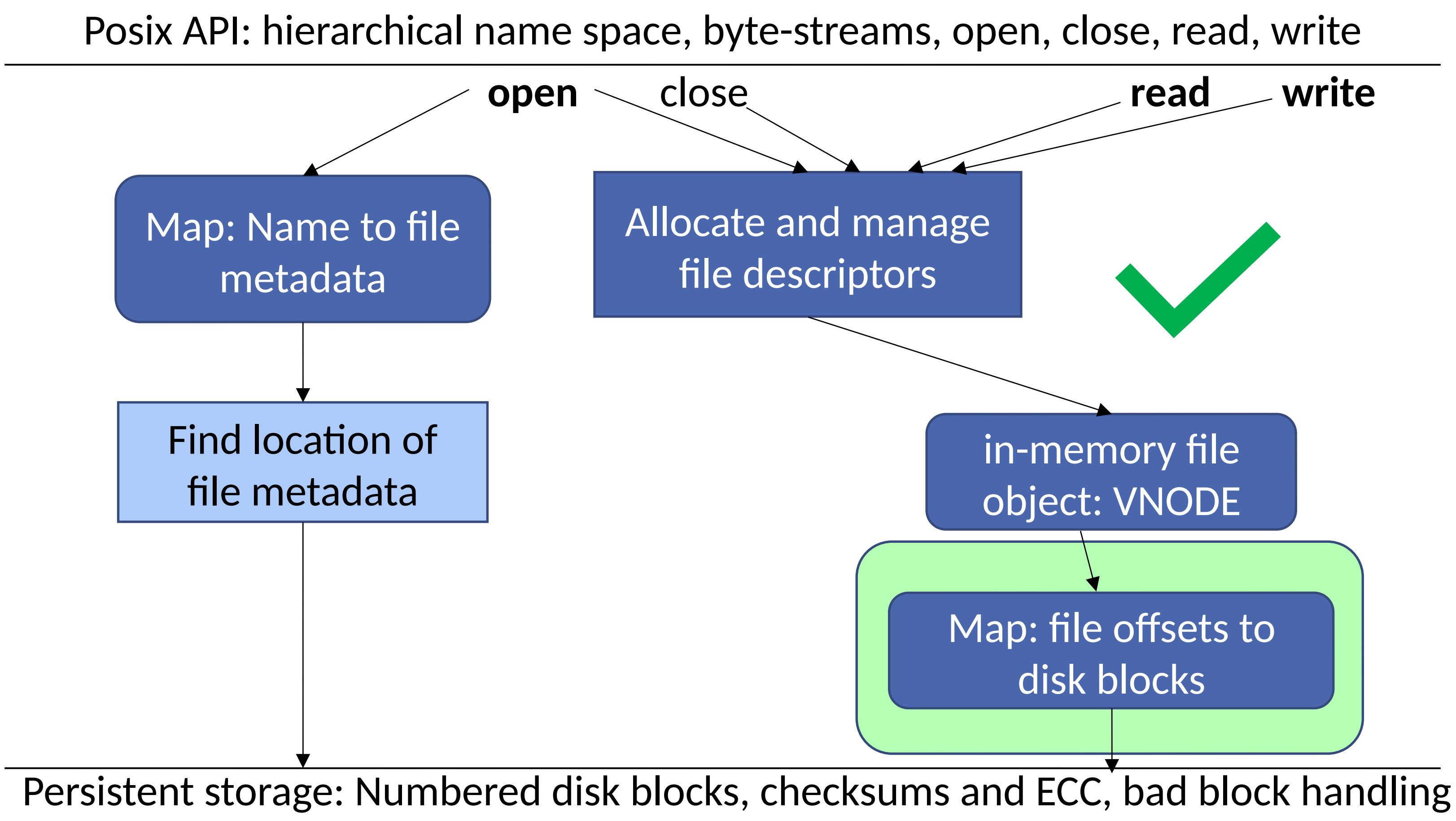
- Next week :
  - No tutorials.
  - No lectures or office hours Monday, Tuesday and Wednesday.

Where we are

- Unit Map:
  - ...
  - P19: File descriptors
  - 4.2. File descriptors management
  - P20: File Systems implementation overview
  - 4.3. How we represent files
  - P21: Why fixed-size block file systems?
  - 4.4. Building a file index

Today

- Today’s Learning Outcomes
  - Explain how files might be represented on disk
  - Use file system terminology:
    - Superblock
    - Inode
    - Inode number or inumber
    - Internal fragmentation
    - External fragmentation
  - Compare and contrast different file representations and identify the tradeoffs among them.



Posix API: hierarchical name space, byte-streams, open, close, read, write
open
close
read
write
Map: Name to file metadata
Allocate and manage file descriptors
Find location of file metadata
in-memory file object: VNODE
Map: file offsets to disk blocks
Persistent storage: Numbered disk blocks, checksums and ECC, bad block handling

File System Design Goals and Constraints

- Long-lived and robust
  - Many files created, deleted, extended and truncated over time
  - Performance should not degrade with time (at least not too much)
- General purpose
  - Different file sizes; files can be sparse (two slides ahead)
  - Different access patterns: sequential and arbitrary (“random”)

File System Design Goals and Constraints

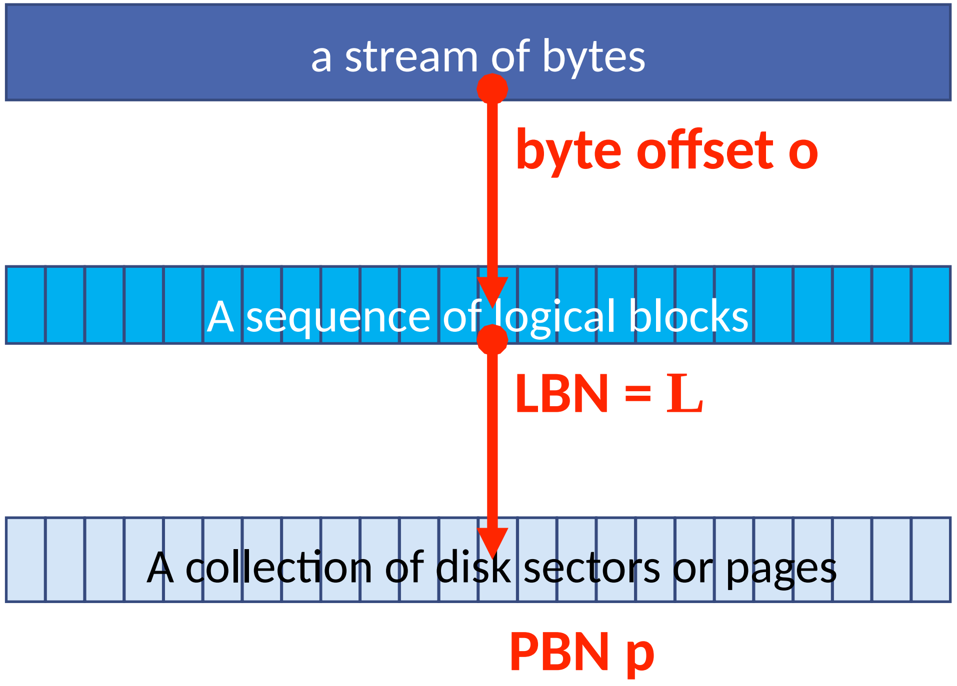
- Performance dictated by storage media hardware (rotating disk or SSD)
  - File is collection of disk blocks that store its data and meta-data
  - Seeking to a block can be much slower than transferring all of its data
  - Where blocks are on the disk can really matter

Sparse Files

- Purpose
  - Some files represent data placed at particular offsets to a starting point…
  - …potentially with large gaps in between…
  - …rather than continuous streams of data.
- Implementation Consideration
  - E.g., Write one byte at offset 0 and another at offset 2^{30} - 1.
    - File's **size** is 2^{30} bytes
    - But, storing that data requires only 2 disk blocks for the file's data
      - Allocating 2^{30} bytes worth of blocks would waste valuable disk space

Layers of Abstraction

- Application
  - Stream of bytes
- File
  - Sequence of Logical Blocks
  - LBN = floor(offset / file_system_block_size)
- Disk
  - Sequence of Physical Blocks
  - A file is a collection of Physical Blocks
  - PBN is computed from LBN using *inode* block map



Basic Data Structures and Concepts

- Super block
  - Meta-data for entire file system.
  - Stored at specific disk locations (e.g., block # 1).
  - To mount a file system OS must be able to read its super block; so, these may be replicated multiple places on disk.

Basic Data Structures and Concepts (cont)

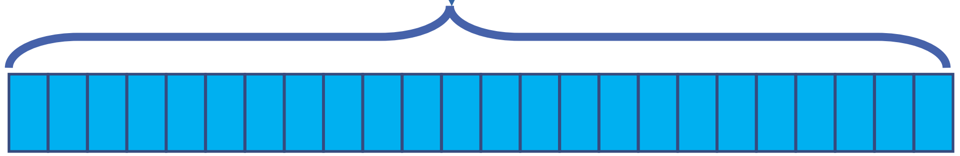
- Inode
  - On-disk meta-data that describes a file
  - Stores: (root of) mapping to disk block # (LBN => PBN) and some other meta-data (file size, file permissions, etc)
  - Does not have a symbolic, human-readable name
- Inumber
  - Internal “name” of an inode
  - File system can map inumber to disk block for that file’s inode
  - ls -i to see file inumbers

Let’s store something in a new file

- Steps
  - **create**: allocate a new inode, thus assigning the file an *inumber*
  - **write**: allocate disk blocks and write data into those blocks
- Questions to ask about how we represent files so we can find the disk block corresponding to each block of data in the file:
  - What would the map from LBN to PBN look like?
  - Does this handle small and large files well?
  - How about sparse files?
  - How well does it handle sequential and random access?

Strategy 1 : Single-Extent-Based Allocation

- Extent
  - Definition: Variable-sized contiguous collection of disk blocks
  - Use: One extent per file; stores all of a file’s data
- LBN to PBN map
  - Simple and small; just store the following two things:
    - Block number of extent’s first block
    - Total number of blocks in the extent
    - Or just the size of the file since we can divide to get the number of blocks

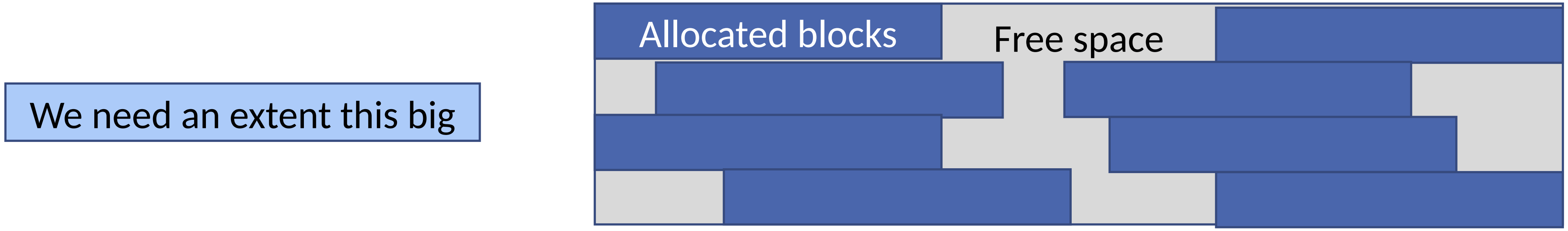


Evaluating Single-Extent-Based Allocation

- Pros
  - Inode is small for all file sizes
  - Sequential access is optimized, matching hardware characteristics
  - Random access needs ≤ 2 reads to get disk data (inode + data block)
- Cons
  - Handles sparse files poorly
  - Does not match file POSIX API
    - When you create a file in Unix you don't tell the OS how big it will be
  - Doesn't handle file extension or truncation well
  - CAUSES EXTERNAL FRAGMENTATION

External Fragmentation (with Single extents)

- Extents are variable-sized, created and deleted *randomly/arbitrarily*
  - Over time, large, contiguous free spaces become scarce
  - They get fragmented into many smaller space



- Even though there's plenty of room for our new file overall, no contiguous free (grey) space is big enough!

NOTES: (1) Recall from 213 that this is the same problem `malloc` implementations face.
(2) Single extents are used for read-only file systems (DVDs, BlueRays).

Strategy 2 : Block-Based Allocation

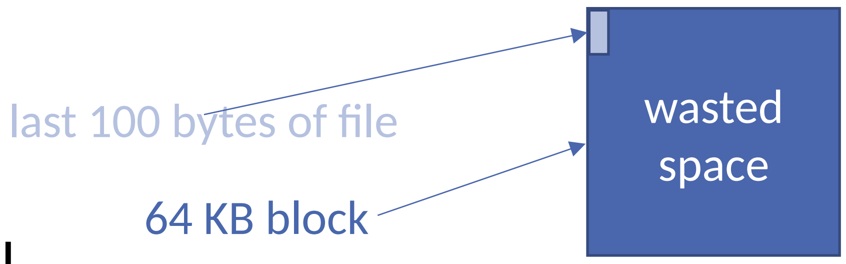
- Blocks
  - Fixed size units of allocation, thus **eliminating external fragmentation**
  - A file might require many blocks
- LBN to PBN map
  - Must store block number of **every block** of the file
  - So, inodes for large files need to be large
    - We'll talk about how to handle this issue next class

Evaluating Block-Based Allocation

- Pros
  - No external fragmentation (if I can fit one more block, I can grow a file by one more block).
  - Matches Unix API; files start out empty.
  - Easy to extend and truncate.
  - Handles sparse files well.
- Cons
  - Not optimized for sequential access
    - Each block may be in a different location on disk
  - Might not be optimized for random access for large files either.

Picking a Block Size

- Pros of big blocks
  - Better performance with sequential accesses (or high spatial locality, generally)
    - Must be at least big enough to amortize fixed access latency (e.g., seek time)
  - Smaller inode block maps
- Cons of big blocks
  - More INTERNAL FRAGMENTATION
    - Last block of file might not be full
  - Particularly problematic for small files …
    - … and lots of files are small



File sizes – Norm's laptop in Fall 2022

- 10,972,857 files

Size	Count	Percent
0 bytes	269,251	2.5
> 0 bytes && <= 4K bytes	6,412,059	58.4
> 4K bytes && <= 8K bytes	1,324,083	12.1
> 8K bytes && <= 16K bytes	1,034,937	9.4
> 16K bytes && <= 1M bytes	1,600,065	14.6
> 1M bytes && <= 1G bytes	332,166	3.0
> 1G bytes	296	0.0

File sizes – Norm’s laptop in Fall 2023

- 12,921,244 files

Size	Count	Percent	Blocks	Blocks Percent
0 bytes	275,553	2.1	0	0.0
> 0 bytes && <= 4K bytes	8,601,494	66.6	8,601,494	0.6
> 4K bytes && <= 8K bytes	1,043,689	8.1	2,087,378	0.2
> 8K bytes && <= 16K bytes	895,783	6.9	3,009,689	0.2
> 16K bytes && <= 1M bytes	1,698,584	13.1	53,202,543	4.0
> 1M bytes && <= 1G bytes	405,437	3.1	807,101,780	60.6
> 1G bytes	704	0.0	457,898,242	34.4

In-class Exercise

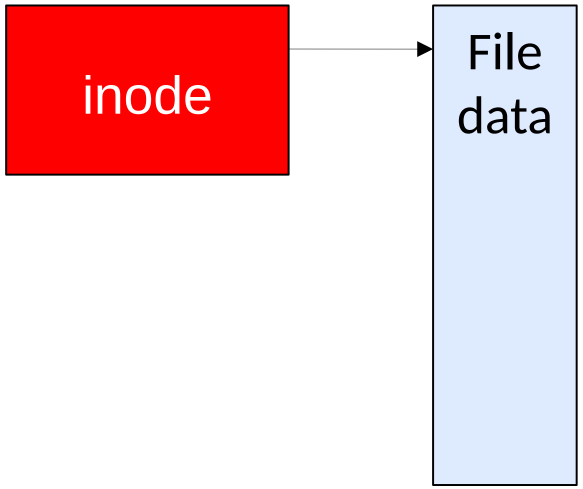
- Practice using knowledge of disks to compute performance of different on-disk allocation strategies.
- While some of the calculations are finicky, the important takeaway is to observe what a dramatic effect layout has on performance.
- Similarly, observing this effect should give you intuition for why solid-state, SSDs (aka flash) drives are so much faster than spinning disks.

Wrapping Up

There are advantages and disadvantages to different layouts and our goal as file system developers* is to pick representations that offer a good set of tradeoffs.

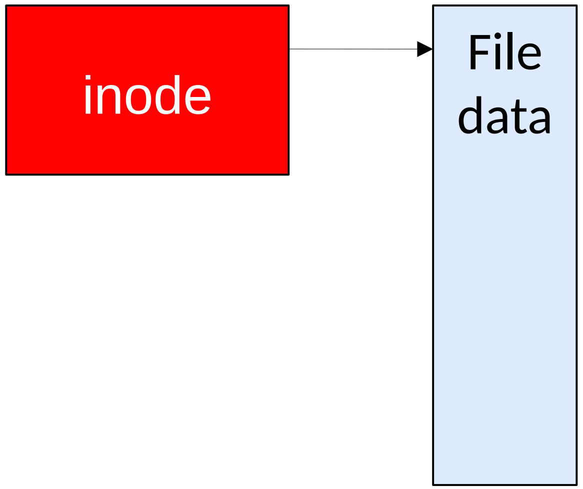
*Note that we used to refer to you all as computer architects; in this unit you are becoming file system designers.

File Representation: **Single Extent**



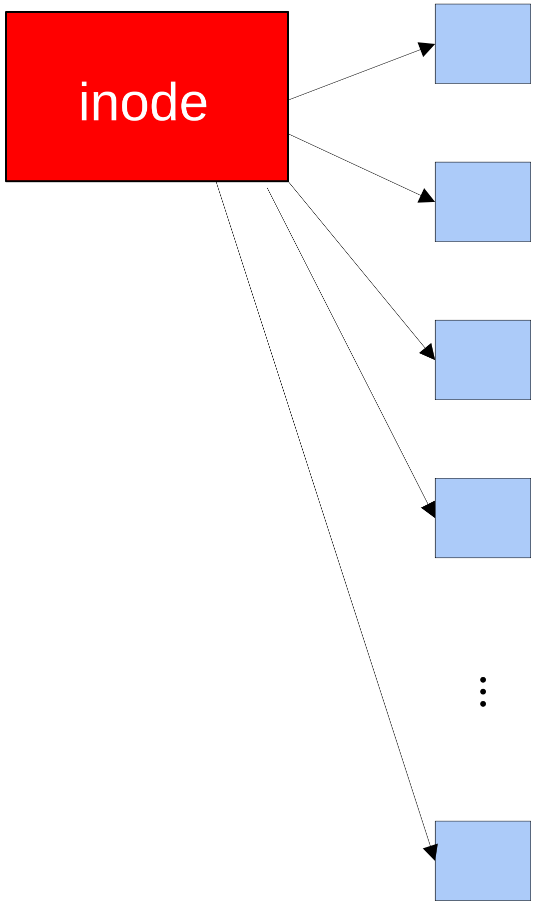
- Pros:
  - really simple
  - good for both sequential and random access
  - very efficient (in terms of memory allocation)
  - relatively little internal fragmentation

File Representation: **Single Extent**



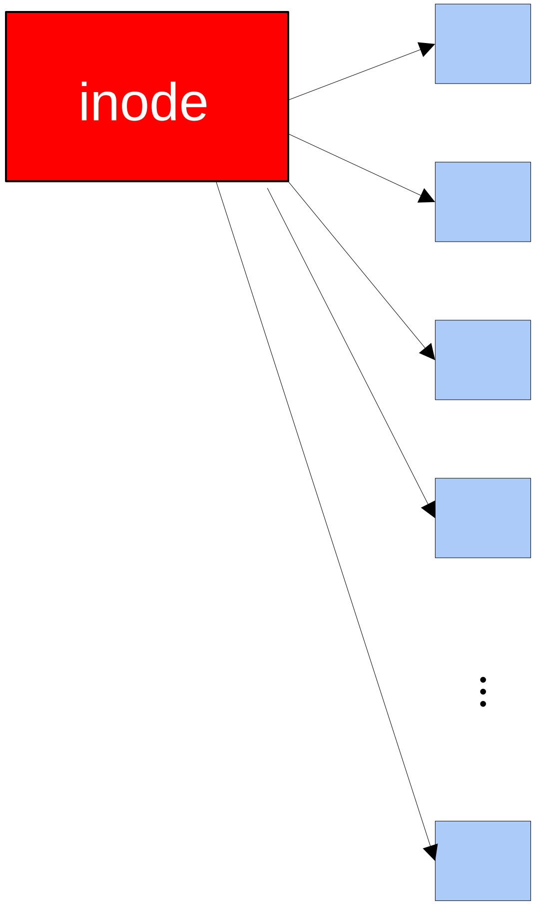
- Cons:
  - inflexible – what happens if a file changes size?
  - have to pre-allocate space at create time
  - lots of external fragmentation
  - wastes space for sparse files
  - Strict single-extent allocation is unrealistic!

File Representation: **Fixed size blocks**



- Pros:
  - eliminates external fragmentation
  - internal fragmentation can be reduced by choosing smaller block sizes in the design
  - easy to grow (and shrink) files
  - handles sparse files well

File Representation: **Fixed size blocks**



- Cons:
  - requires a lot of metadata for big files
    - at least one disk-address per block
  - sequential access could be slow
    - if blocks are scattered over disk
    - each block access could require expensive seek