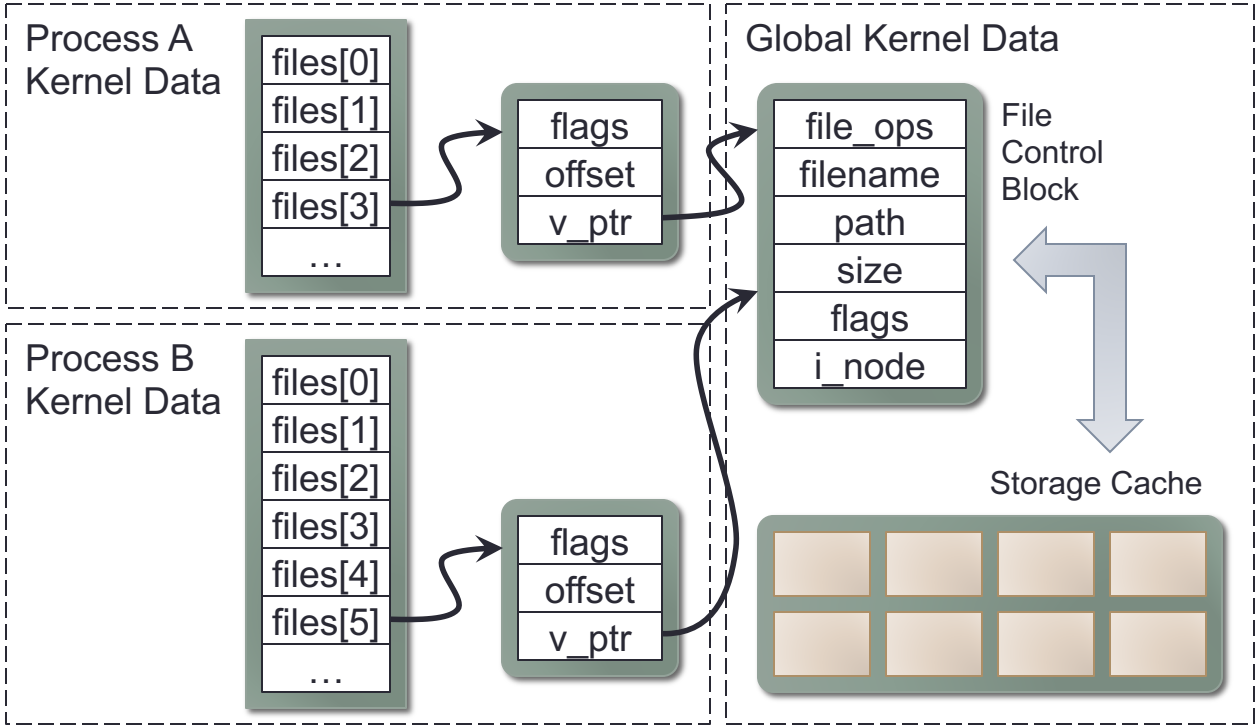
FILE SYSTEMS, PART 2

CS124 – Operating Systems

Spring 2024, Lecture 22

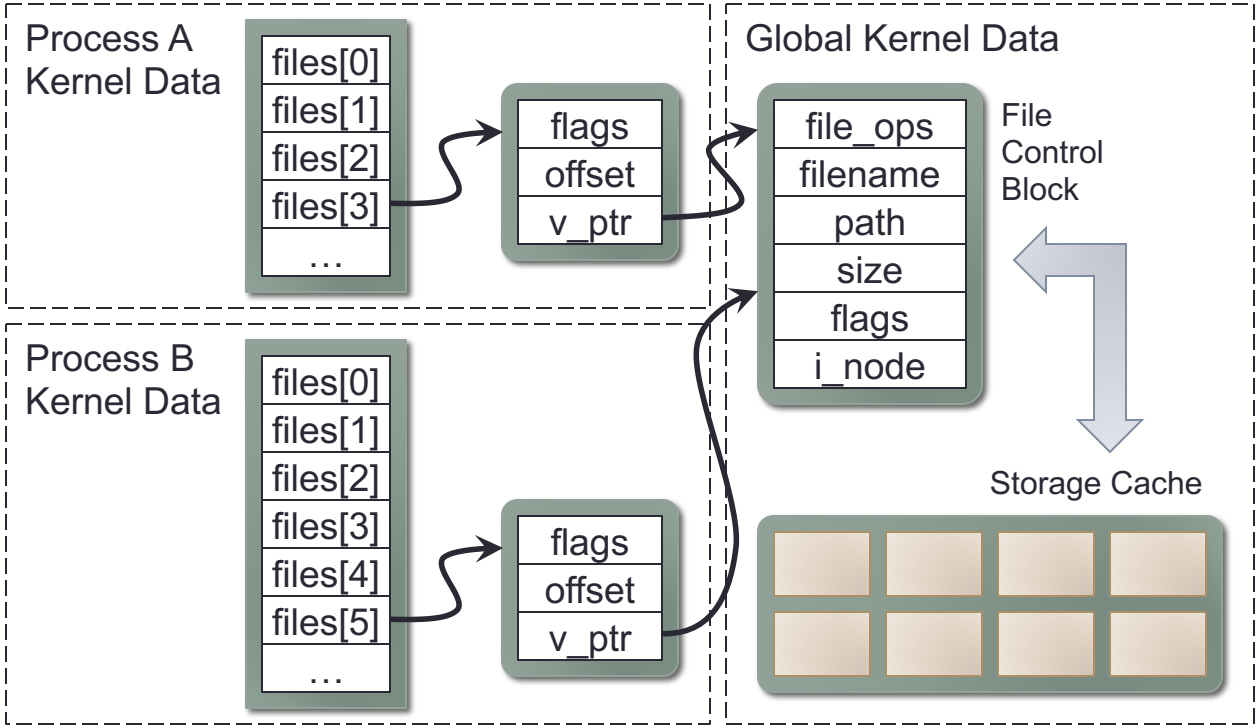
Files and Processes

- The OS maintains a buffer of storage blocks in memory
  - Storage devices are often much slower than the CPU; use caching to improve performance of reads and writes
- Multiple processes can open a file at the same time…



Files and Processes (2)

- Very common to have different processes perform reads and writes on the same open file
- OSes tend to vary in how they handle this circumstance, but standard APIs can manage these interactions



Files and Processes (3)

- Multiple reads on the same file generally never block each other, even for overlapping reads
- Generally, a read that occurs after a write, should reflect the completion of that write operation

- Writes should sometimes block each other, but operating systems vary widely in how they handle this
  - e.g. Linux prevents multiple concurrent writes to the same file
- Most important situation to get correct is appending to file
  - Two operations must be performed: the file's space is extended, then write is performed into newly allocated space
  - If this task isn't atomic, results will likely be completely broken files

Files and Processes (4)

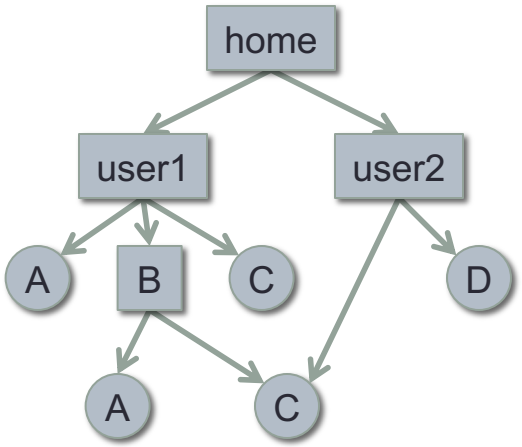
- Operating systems have several ways to govern concurrent file access
- Often, entire files can be locked in shared or exclusive mode
  - e.g. Windows `CreateFile()` allows files to be locked in one of several modes at creation
  - Other processes trying to perform conflicting operations are prevented from doing so by the operating system
- Some OSes provide **advisory file-locking** operations
  - Advisory locks aren't enforced on actual file-IO operations
  - They are only enforced when processes participate in acquiring and releasing these locks
- Example: UNIX `flock()` acquires/releases *advisory* locks on an entire file
  - Processes calling `flock()` will be blocked if a conflicting lock is held…
  - If a process decides to just directly access the `flock()`'d file, the OS won't stop it!

Files and Processes (5)

- Example: UNIX **`lockf()`** function can acquire/release advisory locks on a region of a file
  - i.e. lock a section of the file in a shared or exclusive mode
  - Windows has a similar capability
- Both **`flock()`** and **`lockf()`** are wrappers to **`fcntl()`**
- **`fcntl()`** can perform many different operations on files:
  - Duplicate a file descriptor
  - Get and set control flags on open files
  - Enable or disable various kinds of I/O signals for open files
  - Acquire or release locks on files or ranges of files
  - etc.
- Some OSes also provide **mandatory file-locking** support
  - Processes are forced to abide by the current set of file locks
  - e.g. Linux has mandatory file-locking support, but this is non-standard

File Deletion

- File deletion is a generally straightforward operation
  - Specific implementation details depend heavily on the file system format
- General procedure:
  - Remove the directory entry referencing the file
  - If the file system contains no other hard-links to the file, record that all of the file's blocks are now available for other files to use



- The file system must record what blocks are available for use when files are created or extended
- Often called a **free-space list**, although many different approaches are used to record this information
- Some file systems already have a way of doing this, e.g. FAT formats simply mark clusters as unused in the table

Free Space Management

- A simple approach: a bitmap with one bit per block
  - If a block is free, the corresponding bit is 1
  - If a block is in use, the corresponding bit is 0
- Simple to find an available block, or a run of available blocks
  - Can make more efficient by accessing the bitmap in units of words, skipping over entire words that are 0
- This bitmap clearly occupies a certain amount of space
  - e.g. a 4KiB block can record the state of 32768 blocks, or 128MiB of storage space
  - A 1TB disk would require 8192 blocks (32MiB) to record the disk's free-space bitmap
- The file system can break this bitmap into multiple parts
  - e.g. Ext2 manages a free-block bitmap for groups of blocks, with the constraint that each group's bitmap must always fit into one block

Free Space Management (2)

- Another simple approach: a linked list of free blocks
  - The file system records the first block in the free list
  - Each free block holds a pointer to the next block
- Also very simple to find an available block
  - Much harder to find a run of contiguous blocks that are available
- Tends to be more I/O costly than the bitmap approach
  - Requires additional disk accesses to scan and update the free-list of blocks
  - Also, wastes a lot of space in the free list…
- A better use of free blocks: store the addresses of many free blocks in each block of the linked list
  - Only a subset of the free blocks are required for this information
- Still generally requires more space than bitmap approach

Free Space Management (3)

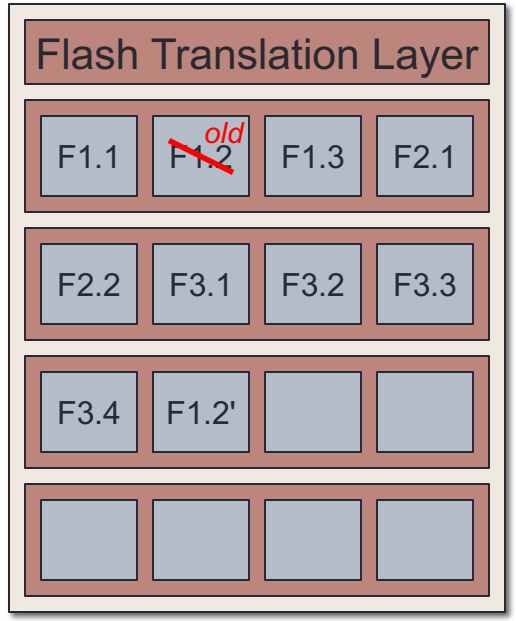
- Many other ways of recording free storage space
  - e.g. record runs of free contiguous blocks with (start, count) values
  - e.g. maintain more sophisticated maps of free space
- A common theme: when deleting a file, many of these approaches don't actually require touching the newly deallocated blocks
  - e.g. update a bitmap, store a block-pointer in another block, …
- Storage devices usually still contain the old contents of truncated/deleted files
  - Called **data remanence**
- Sometimes this is useful for data recovery
  - e.g. file-undelete utilities, or computer forensics when investigating crimes
- (Also generally not difficult to securely erase devices)

Free Space and SSDs

- Solid State Drives (SSDs) and other flash-based devices often complicate management of free space
- SSDs are block devices; reads and writes are a fixed size
- Problem: can only write to a block that is currently empty
- Blocks can only be erased in groups, not individually
  - An **erase block** is a group of blocks that are erased together
  - This is done primarily for performance reasons
- Erase blocks are much larger than read/write blocks
  - A read/write block might be 4KiB or 8KiB…
  - Erase blocks are often 128 or 256 of these blocks (e.g. 2MiB)
- As long as some blocks on SSD are empty, data can be written immediately
- If the SSD has no more empty blocks, a group of blocks must be erased to provide more empty blocks

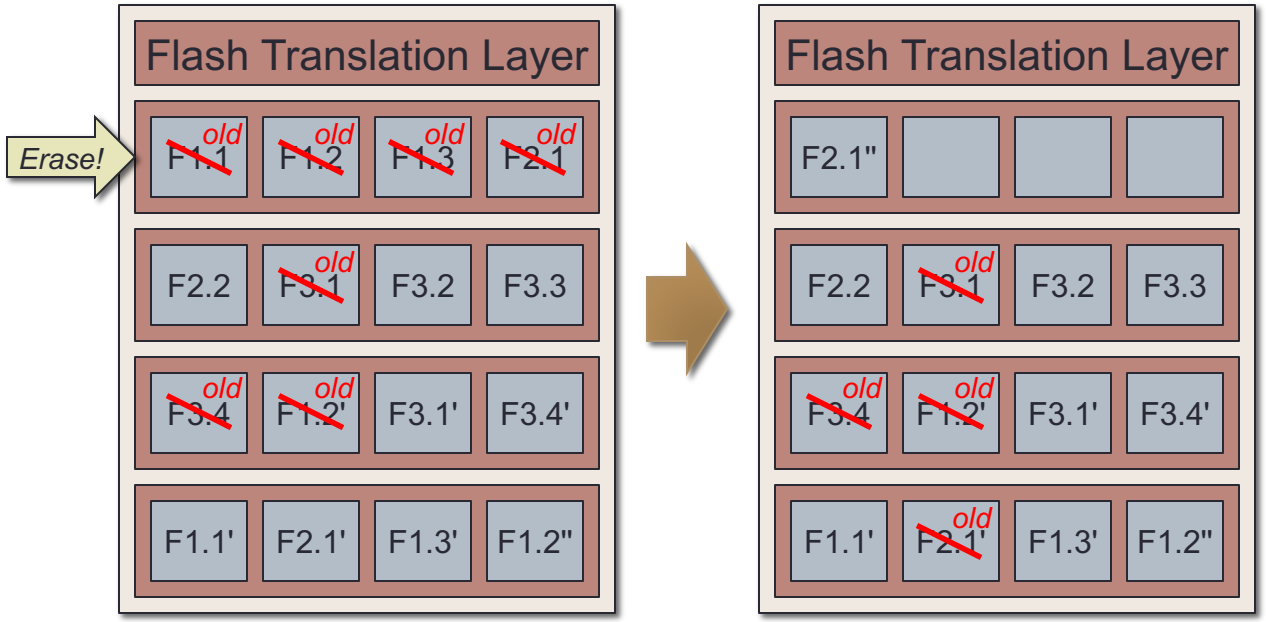
Solid State Drives

- Solid State Drives include a **flash translation layer** that maps logical block addresses to physical memory cells
  - Recall: system uses Logical Block Addressing to access disks
- When files are written to the SSD, data must be stored in empty cells (i.e. old contents can't simply be overwritten)
- If a file is edited, the SSD sees a write issued against the same logical block
  - e.g. block 2 in file F1 is written
- SSD can't just replace block's contents…
- SSD marks the cell as "old," then stores the new data in another cell, and updates the mapping in the FTL



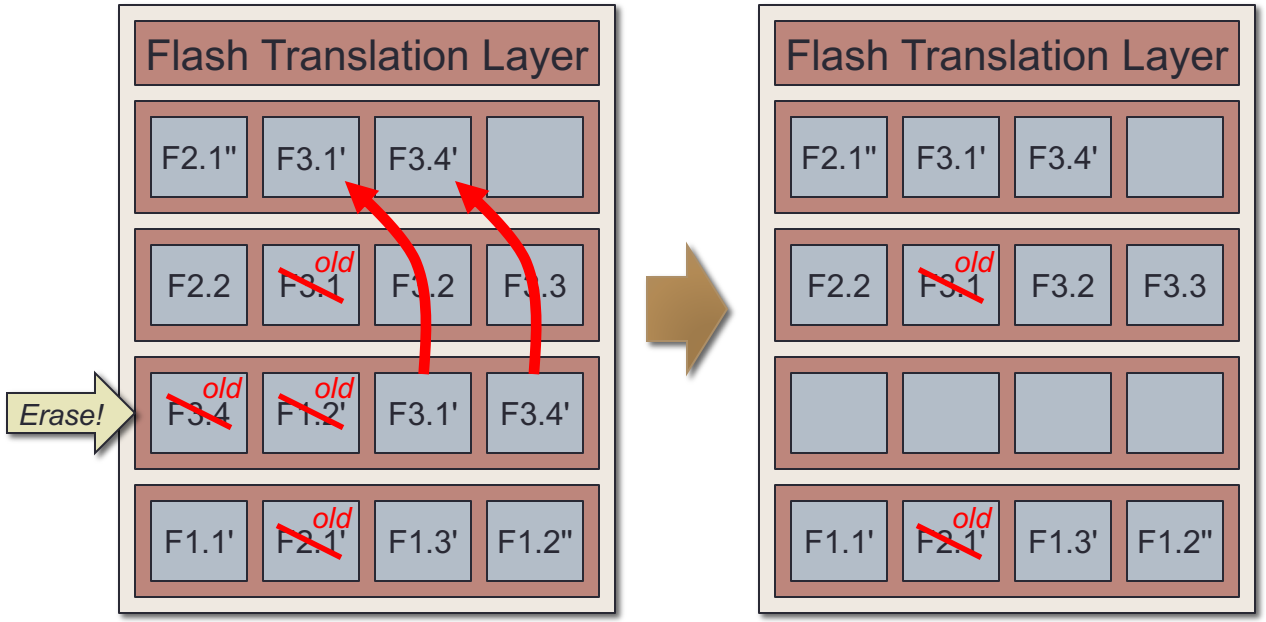
Solid State Drives (2)

- Over time, SSD ends up with few or no available cells
  - e.g. a series of writes to our SSD that results in all cells being used
- SSD must erase at least one block of cells to be reused
- Best case is when an entire erase-block can be reclaimed
  - SSD erases the entire block, and then carries on as before



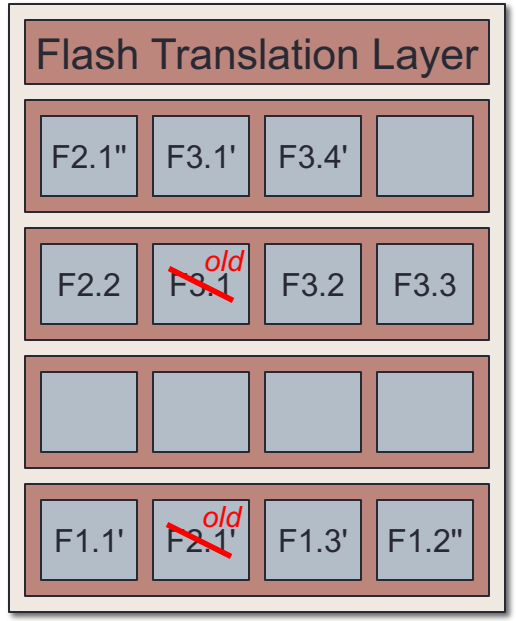
Solid State Drives (3)

- More complicated when an erase block still holds data
  - e.g. SSD decides it must reclaim the third erase-block
- SSD must relocate the current contents before erasing
- Result: sometimes a write *to* the SSD incurs additional writes *within* the SSD
  - Phenomenon is called **write amplification**



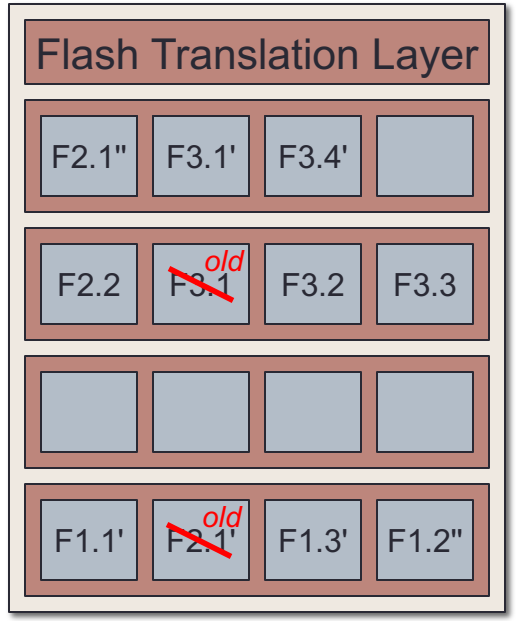
Solid State Drives (4)

- SSDs must carefully manage this process to avoid uneven wear of its memory cells
  - Cells can only survive so many erase cycles, then they become useless
  - Technique is called **wear leveling**
- How does the SSD know when a cell's contents are no longer needed? (i.e. when to mark the cell "old")
- The SSD only knows because it sees several writes to the same logical block
  - The new version replaces the old version, so the old cell is no longer used for storage



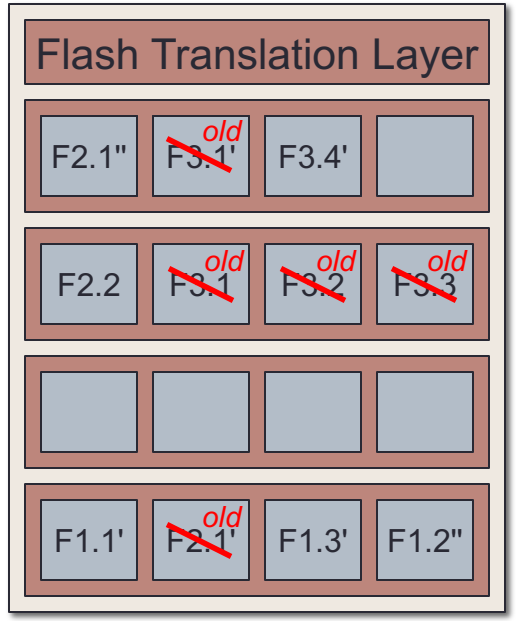
SSDs and File Deletion

- Problem: for most file system formats, file deletion doesn't actually touch the blocks in the file themselves!
  - File systems try to avoid this anyway, because storage I/O is slow!
  - Want to update the directory entry and the free-space map only, and want this to be as efficient as possible
- Example: File F3 is deleted from the SSD
  - The SSD will only see the block with the directory entry change, and block(s) holding the free map
- The SSD has no idea that file F3's data no longer needs to be preserved
  - e.g. if the SSD decides to erase bank 2, it will still move F3.2 and F3.3 to other cells, even though the OS and the users don't care!



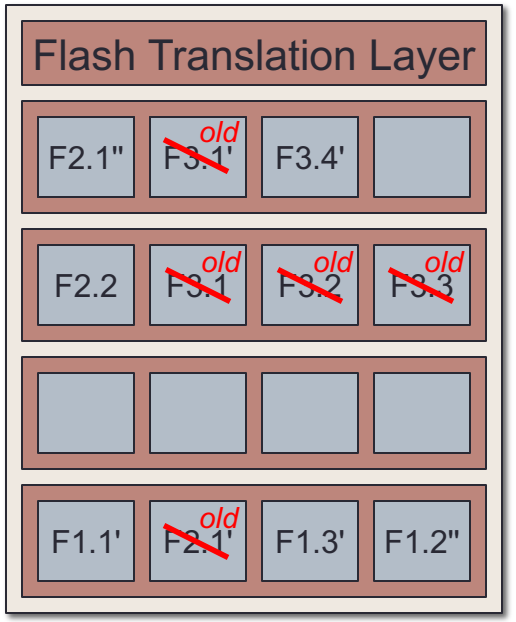
SSDs, File Deletion and TRIM

- To deal with this, SSDs introduced the TRIM command
  - (TRIM is not an acronym)
- When the filesystem is finished with certain logical blocks, it can issue a TRIM command to inform the SSD that the data in those blocks can be discarded
- Previous example: file F3 is deleted
  - The OS can issue a TRIM command to inform SSD that all associated blocks are now unused
- TRIM allows the SSD to manage its cells much more efficiently
  - Greatly reduces write magnification issues
  - Helps reduce wear on SSD memory cells



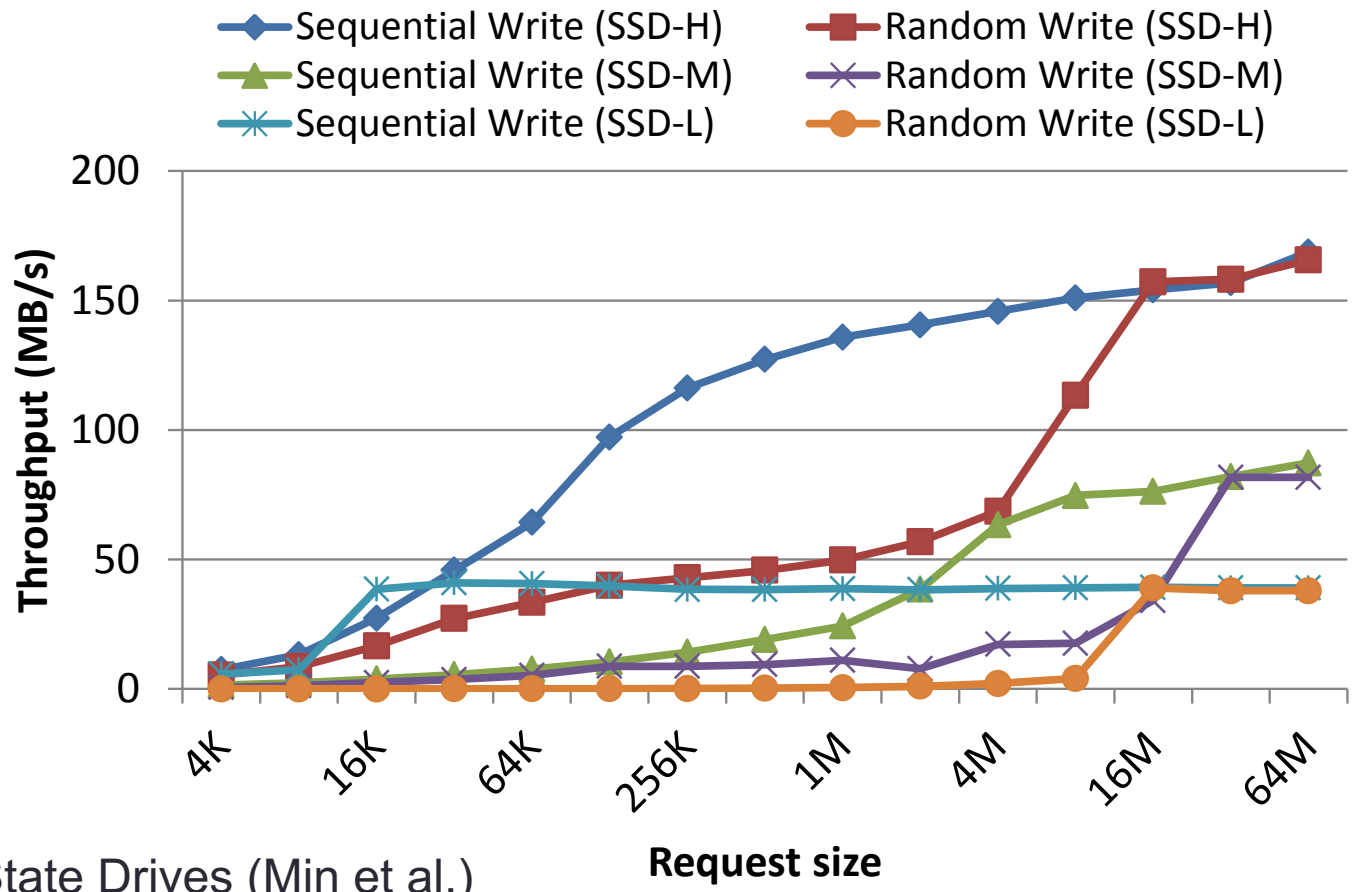
SSDs and Random Access

- A common claim about SSDs is that random access is the same performance as sequential access
  - The Flash Translation Layer is solid-state logic
  - No mechanical devices that must move over a distance
- Really only true for random vs. sequential reads
- Depending on size of writes being performed, random write performance can be much slower than sequential writes
- Reason:
  - Small random writes are much more likely to be spread across many erase blocks…
  - Random writes are likely to vary widely in when they can be discarded…
  - **Overhead of write amplification is increased in these scenarios**
- Sequential writes tend to avoid these characteristics, so overhead due to write amplification is reduced



SSDs and Random Access

- If random-write block size grows to the point that it works well with the SSD erase-block size and garbage collection algorithm, then random writes will be as fast as sequential writes
- Below that size, sequential writes to an SSD are much faster than random writes
- This affects the design of SSD-friendly filesystems and database file layouts



SFS: Random Write Considered Harmful in Solid State Drives (Min et al.)

Next Time

- Next time: notes on the Pintos filesystem assignment