CS647 Modern Storage Systems

- Data recovery
- In-place/log
- Remap/shadowing

Angelos Bilas, bilas@ics.forth.gr

CS647 Modern Storage Systems, Fall 2022
(Modern Key-Value Store Design and Implementation)
Angelos Bilas, bilas@csd.uoc.gr
Computer Science Department, University of Crete

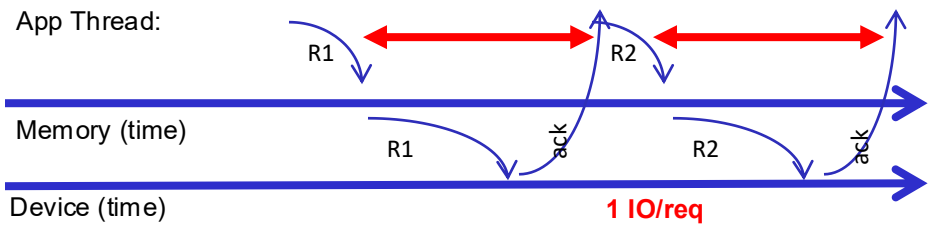
Error Detection & Failure Recovery

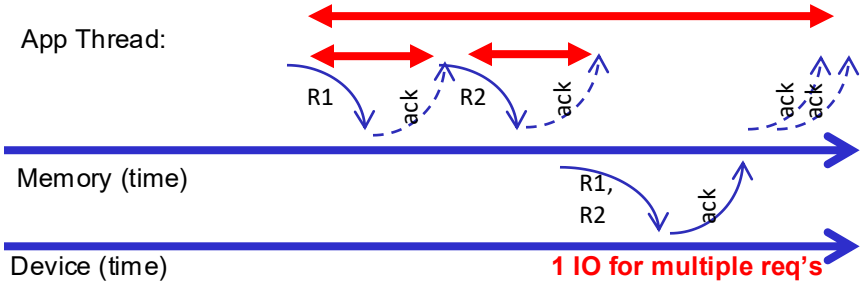
- All storage systems need to (1) detect block errors and (2) recover from failures
- Device checksums are necessary but not sufficient
  - EDC checksum: error detecting code
  - Computed on writes, computed and checked on reads
  - Based on above, "ok" does not mean "everything is ok"
  - Checksums help detect corrupted blocks, but we lose the data in corrupted blocks
  - ECC checksum: error correcting codes – correct a few bits in a FLASH page
- Types of device/block errors, e.g. during a power failure, silent, etc.
  - Zheng et al. Reliability Analysis of SSDs Under Power Fault. ACM Trans. Comput. Syst. 34 (Jan. 2017). https://doi.org/10.1145/2992782
  - These are errors that are "expected" to happen in a system
    - Silent bit corruption: A bit flips for some reason (also cases of silent corruption during lifetime of data)
    - Read disturb (SSDs): Reading a block many times can cause neighboring blocks to change value
    - Incomplete (shorn/torn) writes: Writes that complete partially, corrupted sectors/blocks/IOs
      - Shorn = sector/block size, Torn = multi-block size
      - Failure atomicity: both multi-block and single-block write IOs are atomic
      - Concurrency atomicity: single-block writes are atomic, multi-block writes are not atomic
    - Dead device: A device does not respond
    - Most (except torn writes) can be detected with checksums in data blocks
  - These are errors "not expected", e.g. hw/sw bugs in device
    - Flying/misplaced+lost writes: Writes complete but placed in wrong block (e.g. metadata error)
    - Device metadata corruption in the device: E.g. bit corruption in FTL metadata
    - Serialization (ordering): A set of operations are written to the device but the order is wrong
    - Cannot be detected with checksums in data blocks
- What does a storage system need to do?
  - 1. Catch as many errors of the above as possible (compared to relying only on device checksums)
  - 2. Recover from failures: Deal with corrupted/wrong blocks after a failure is detected
    - Distinguish between metadata and data blocks
- All storage systems need to do (2) and some do more for (1) than others

Recovery

- What does it mean to recover from failures? → Crash recovery
  - When re-starting from an uncontrolled failure (=crash), blocks (data/metadata) must be
    - correct = no corrupted bits/data in blocks – Sometimes filesystems allow corruption for data blocks
      - Cannot read data that were not written
    - consistent = order of user writes is preserved, metadata + data must agree
      - Cannot read recent data and lose older data, leak space, or fail valid operations
    - last-write persistent = each block must have the last value acknowledged to the app
      - Should always find the latest data written by app
  - Recover from both host and device failures (and network too if distributed)
- Basic concept – To achieve recovery we cannot do in-place updates
  - Recovery needs a 2nd (transient) copy for each write
  - Failures can always corrupt a block during a write
  - Then we lose both the old and the new value
  - We need to maintain somewhere a second, transient copy
  - Once we write a new value properly, we can reclaim one of the two copies
- Two different approaches to manage the 2nd copy
  - L.F. Cabrera, J. A. McPherson, P.M. Schwarz and J. C. Wyllie, "Implementing atomicity in two systems: techniques, tradeoffs, and experience," in *IEEE Transactions on Software Engineering*, vol. 19, no. 10, pp. 950-961, Oct. 1993, https://dl.acm.org/doi/10.1109/32.245737
  - 1. Log-based Recovery: Write new block in a log, **copy new block in-place**, free the log
  - 2. Shadow Recovery: Write new block elsewhere, **remap to new block**, remove old block
- Notice that (1) has two writes, whereas (2) has a single write
  - Still: (1) is used more extensively today
  - Both approaches significant complexity

Group commit & Delayed application acks





- What does it mean to recover?
  - When re-starting from an uncontrolled failure (=crash), data must be consistent ~~**AND** each data item must have the last value acknowledged to the client~~ (either last-write or prefix)
  - What does "recovery" guarantee?
    - R. Hagmann. 1987. Reimplementing the Cedar file system using logging and group commit. In Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87). Association for Computing Machinery, New York, NY, USA, 155–162. https://doi.org/10.1145/41457.37518
    - T.S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. 2017. Application crash consistency and performance with CCFS. In Proc. of the 15th USENIX Conf. on File and Storage Technologies (FAST'17). https://www.usenix.org/conference/fast17/technical-sessions/presentation/pillai
- (1**) For each request,** write to the device then ack user → **"last write"** recovery (application consistency)
- (2) **Batch a number of requests → "group commit"** recovery
  - Choice 1: ack immediately → low latency but no last-write recovery = **prefix recovery**
    - After a failure the system recovers to a previous point, where if a write is committed then all previous writes (in Program Order) are committed as well
    - All subsequent writes are dropped
    - Writes can be acknowledged to the application when they arrive to the system, before being written to the device
  - Choice 2: ack after persist → **last-write recovery** but (very) long latency for response to user
- Each thread issues synchronous requests (program order)
  - If asynchronous within a thread, then writes are concurrent and any order is fine
  - Requests from multiple threads are asynchronous (concurrent, unordered)
- Group commit = batch then persist
  - Only way to avoid per-IO synchronous writes to the device
  - May or may not delay acks to applications, based on required recovery (last-write or prefix)
  - Today, most KV stores, do not delay, otherwise latency too long (so prefix recovery)
- Systems can support both last-write (with or without group commit) or prefix recovery

Log-based Recovery in KV stores

- Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The Recovery Manager of the System R Database Manager. ACM Comput. Surv. 13, 2 (June 1981), 223–242. https://doi.org/10.1145/356842.356847
- R. Hagmann. 1987. Reimplementing the Cedar file system using logging and group commit. In Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87). Association for Computing Machinery, New York, NY, USA, 155–162. https://doi.org/10.1145/41457.37518
- C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. **ARIES:** a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. 17, 1 (March 1992), 94–162. https://doi.org/10.1145/128765.128770
- Goetz Graefe. 2012. A survey of B-tree logging and recovery techniques. ACM Trans. Database Syst. 37, 1, Article 1 (February 2012), 35 pages. https://doi.org/10.1145/2109196.2109197
- Russell Sears and Eric Brewer. 2009. Segment-based recovery: write-ahead logging revisited. Proc. VLDB Endow. 2, 1 (August 2009), 490–501. https://doi.org/10.14778/1687627.1687683

- Log → Write Ahead Log (WAL)
  - Many variants depending on system semantics, requirements, design, etc.
  - Database logging, filesystem logging, KV store logging, e.g.
  - DBs use logging for ACID: atomicity, consistency, isolation, and durability (steal/no-force approach)
  - FSes and KV stores, mostly for consistency and durability for metadata (no-steal/no-force approach)
- For this discussion, limit scope to single node
  - Multiple threads, issuing synch requests (async similar)
  - Networked case is similar when each KV pair is served by a single server
    - aka: scale-out, sharded, partitioned, etc.
- WAL basics
  - Write every operation to a log in program order
  - Ack to application, then start updating in-place
  - Achieve this with low overhead (latency, xput, cpu, IO, mem)
    - FSes, KV stores offer durability and consistency
    - They use REDO log operations, but avoid UNDO operations

WAL Issue (failure free) Path

- 1. KV ops (assume write/update -- reads, scans do not matter) arrive to KV store from many threads
  - Scans can be complicated if they need to access a consistent view of the data without updates in between
  - Batched reads/writes can complicate things further, creating some form of transactions
- 2. Each KV op is written to **a single WAL buffer/segment in memory** which consists of pages/blocks
  - **WAL buffer is linked to current L0 (memory metadata only, no I/O)**
  - **Single WAL buffer** = program order of issuing thread and global order for all threads
- 3. System acks request to application
  - This can be delayed until after device write if last-write recovery
  - The system can ack one or many reqs to each thread, but usually acks individual requests to avoid delays
- 4. After logging in memory, system performs operations to L_0 (in memory)
  - **Order in WAL specifies also arrival order in L_0**
  - Nothing is persistent so far
  - Several threads in parallel, **L_0 does not preserve program order (instead, key order)**
- 5. When WAL buffer fills up
  - 5a. Allocate new WAL buffer in memory and start using it
    - **If no memory, block until some WAL buffer is available**
    - Ultimately average write rate is limited by sequential device write throughput (WAL + compactions)
  - 5b. Issue the new full WAL buffer to the device **asynchronously**
    - We need to be able to identify valid WAL buffers on the device and their write order to guarantee correct kv op order after recovery
  - 5c-A. When device completes WAL IO, **update a superblock with the new segment**
    - Superblock essentially points to valid WAL buffers and their order
    - This approach allows WAL segments to reside anywhere on device, not on a fixed location
    - **Requires 2 IOs/WAL segment: WAL + superblock**
  - 5c-B. The log is located in a fixed area on device **AND** each WAL buffer consists of pages with a header (with logical seqno)
    - Page is the unit of atomic write for the device (sector or block, guaranteed to detect errors with checksum at read time)
    - Fixed area provides ordering information, headers say which buffers are valid
    - Recovery starts from log page 0 and scans to the end, for all log pages that are valid
    - Invalid log pages will have an invalid header/LSN
    - **Requires 1 IO/WAL segment**
  - Both 5c-A and 5c-B allow detecting properly a half-full or corrupted log segment/page
  - **User inserts stall/block only**
    - **When there is no memory left (a) for a new WAL buffer OR (b) L_0 is full and need to be compacted**
    - L_0 compactions have to wait for WAL writes as well before they are committed, to avoid losing writes
- 6. Done

WAL Garbage Collection at L_0 Compaction

- At some point L_0 (memory) will be compacted to L_1 (device)
  - When L_0 to L_1 compaction completes
  - Inserted KV pairs are stable in L_1 on the device (this is a checkpoint)
  - Ordering information for kv requests becomes part of the checkpoint
- WAL is not needed any more and can be garbage collected
  - WAL GC is straight-forward
  - All WAL segments linked to L_0 (which has been compacted) are **trimmed**
  - Trim means we free WAL segments **in order**, without creating "holes"
  - If 5c-A, update superblock with freed segments (1 IO for each L_0 compaction)
  - If 5c-B, simply reuse pages/blocks (new LSNs will be higher than old LSNs, no IO)
- A "proper" log is supposed to use only append + trim segment
  - Alternative WALs could free segments out of order
  - Requires updating a pointer from the previous segment to the next
  - (bypassing the segment that is freed)
  - Additional write to the device for each free operation
- During compaction, application requests need to stall if L_0 full
  - This is a compaction issue, not a WAL-related recovery issue

WAL-based Recovery Cost

- No delay for application acks, no blocking for new ops until L_0 is full
  - Assuming enough WAL buffers (how many?)
- Several WAL buffers require breaking a log to multiple segments
  - Multiple outstanding writes to the device
  - Easy for different regions in the KV store (regions are independent/different logs)
  - More attention for the same KV region
  - Note: using multiple independent logs is not the same as a single log (no global order)
- WAL cost
  - L_0 holds ~1M KV pairs (~100MB), WAL segment holds 10K kv pairs (~1MB) pairs
    - L_0 "is worth" 100 WAL segments
  - (1) Throughput cost is 2x writes, all IOPS are large
    - Except for superblock I/Os (5c-A) below
  - (2) IOPS cost is, every ~10K kv inserts:
    - 5c-A: 1 large IO (~1MB) + 1 small IO (superblock, 4KB) for each WAL segment
      - +1 IO to free WAL segments after compaction (every 1M inserts)
    - 5c-B: 1 large IO (~1MB) for each WAL segment
- Can the ingest path in KVS operate at device throughput?
  - 10s GBytes/s for a couple of NVMe's
  - Best case would be max/2 – for a well-designed log, this is achievable
  - Compaction is significantly higher overhead

Recovery Path

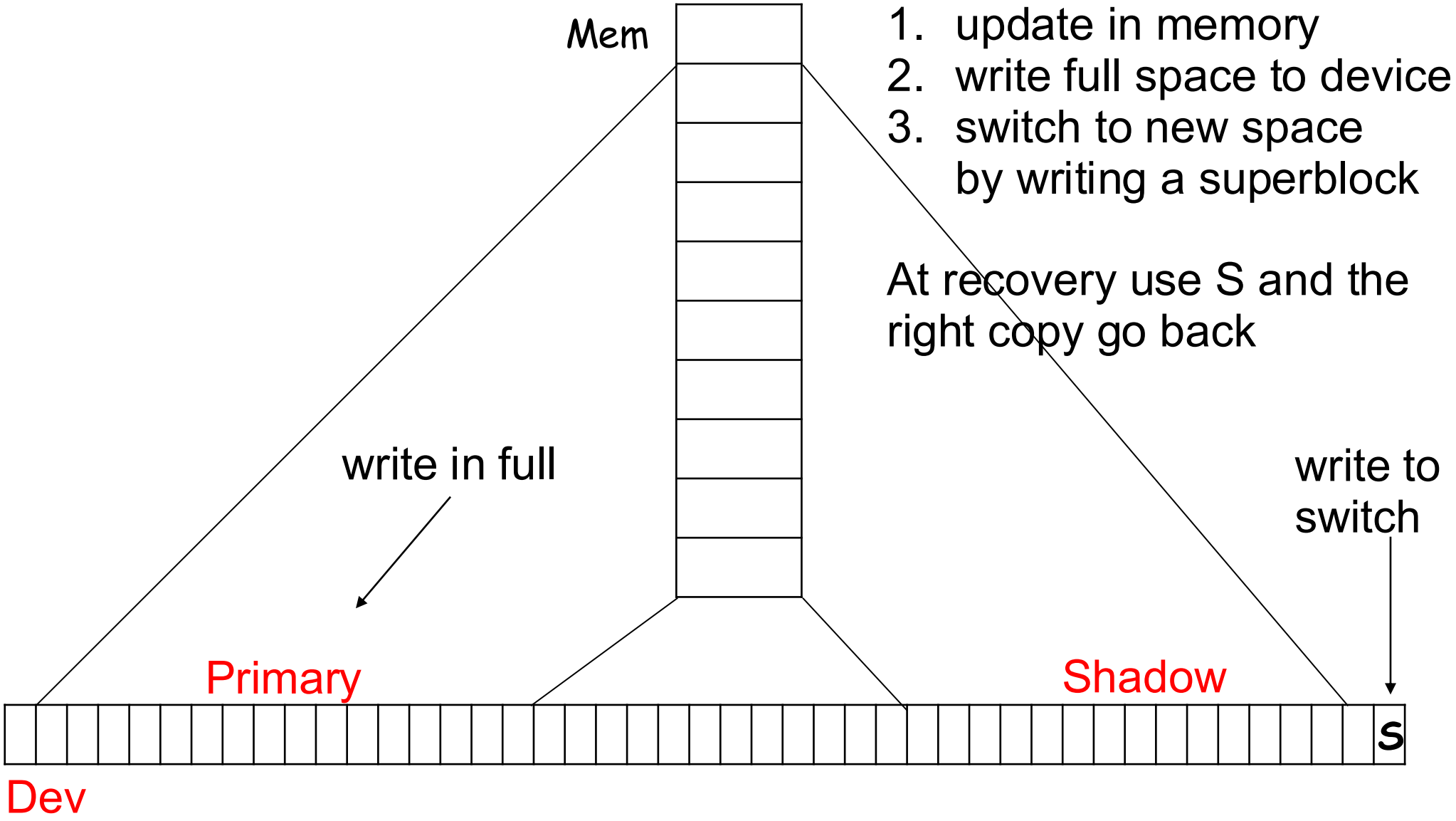
- Relatively simple
- 1. Start the KV store, read superblock and log start on device
- 2. Scan the log from start (and all valid pages/segments) and replay each entry (REDO)
  - Scanning of 100 MB log takes 0.1s at 1 GB/s device speed
  - Entries are replayed in memory (and rebuild L_0)
  - There is no need to UNDO in the KV store levels
- 3. While doing so, L_0 compactions may occur
  - Compaction of 100MB would be another 0.2s: R+W at e.g. 1 GB/s
  - Each compaction will also truncate the log
- 4. Start serving application requests
  - A few secs for several KV regions and 10s GB of log

Remap-based Recovery (Shadowing Concept)

Ohad Rodeh. 2008. B-trees, shadowing, and clones. ACM Trans. Storage 3, 4, Article 2 (Feb. 2008), 27 pages. https://doi.org/10.1145/1326542.1326544
Joost Verhofstad. 1977. Recovery and crash resistance in a filing system. ACM SIGMOD 1977. pp. 158–167. https://doi.org/10.1145/509404.509430

- Consider a set of blocks that we need to read/write and has to be recoverable
- 1. Create two copies of the data on the device – let's say divide the device in half
  - One is the primary area (Dp), the other is the shadow area (Ds)
- 2. Updates to data will arrive in memory – consider a large buffer M with all data/metadata
  - Everything that needs to be recoverable is part of M (but M may contain other stuff as well…)
  - A data update usually requires updating other pointers in M that also have to be persistent
    - Consider a tree (path to root) or a graph with many nodes pointing to a node that gets updated
  - M batches new writes/updates for data and metadata
- 3. At a "checkpoint" interval (e.g. a few secs or few updates), write M to Ds
  - If a failure occurs, Ds will be corrupted, but Dp will contain a consistent version of M for recovery
- 4. After write(M) completes, switch the role of primary and shadow areas on the device
  - This switch needs to be persistent, e.g. by writing a superblock, so we recover from the correct area
  - Let's say that this is a single pointer update recorded in a metadata superblock
- Cost
  - (1) Data (metadata) gets written only once to the device
  - (2) Primary – shadow switch requires at least one additional write to a known location on the device
    - Pointer update needs to happen infrequently → batching (but always random IO)

Remap-based Recovery



Challenges with Remap-based Recovery

- All these can be done ”easily” at the application-level

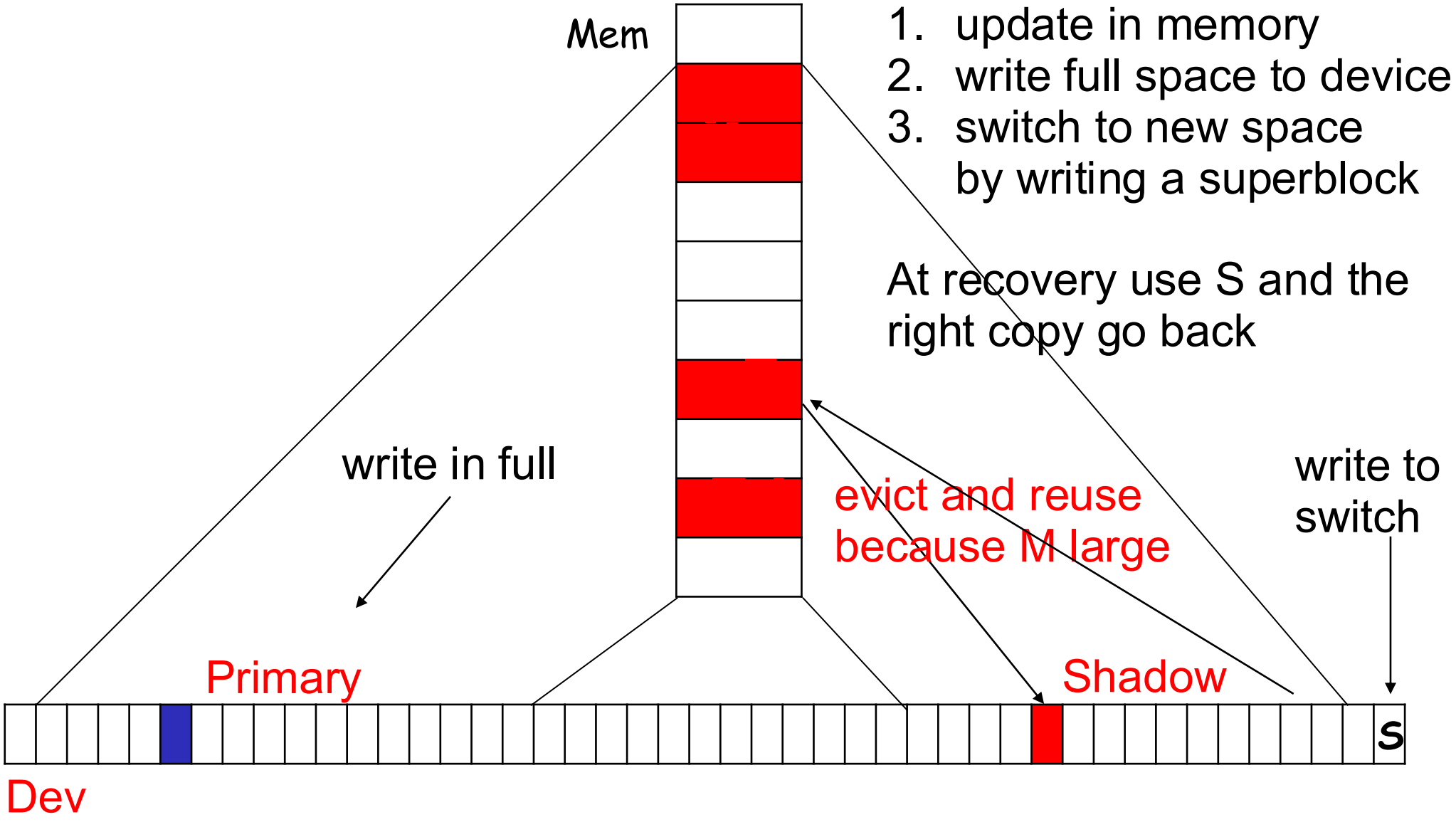
However:

- 1. What if M contains stuff that do not need to be persisted
  - Or everything that needs to be persistent does not nicely fit in a single region of the address space, e.g. a B+-Tree index and allocator metadata
- 2. What if M does not fit in memory – actually the contents of M are larger than available memory
  - Virtual vs. physical address space?
- 3. How frequent do we checkpoint?
  - We need to batch several updates otherwise pointer switch too frequent/expensive
- 4. Until checkpoint is done, we cannot write M (why?)
  - We may break order of updates
- 1,2,3,4 can be significant problems

Remap with Memory Mapped I/O

- Memory-mapped I/O helps with problem 1 and 2
  - What is memory-mapped I/O?
  - What is file-backed memory in the kernel?
- Consider M is virtual address space, mapped to Ds (and Dp is stable)
  - Mechanism exists in kernel for file-backed memory (single mapping, but we can extend)
  - M can be as large as it needs and the kernel will take care of caching and evictions to Ds
- Allow a number (one or many) of updates to M as before
- At a checkpoint interval flush M (to Ds)
  - It will write to the device **only dirty (written) pages** of M that are in memory – msync()
  - The rest are already clean in Ds, no need to rewrite
  - But Dp, Ds may differ in other pages as well, that were evicted earlier to the device
- After the checkpoint is done we need to make Ds the new stable file
  - Ds is the most up-to-date copy of M - if we switch to Dp, it will contain old data for M
  - Need to **copy data from Ds to Dp** and then switch M to be backed by Dp
  - Result of M (memory) not containing all data persisted in Ds, Dp
- Then, switch roles of Ds and Dp as before and allow application to continue
- So with mmio M can be large and we write only dirty pages to the device
- BUT: We introduce a new overhead, synching Dp, Ds before switching roles

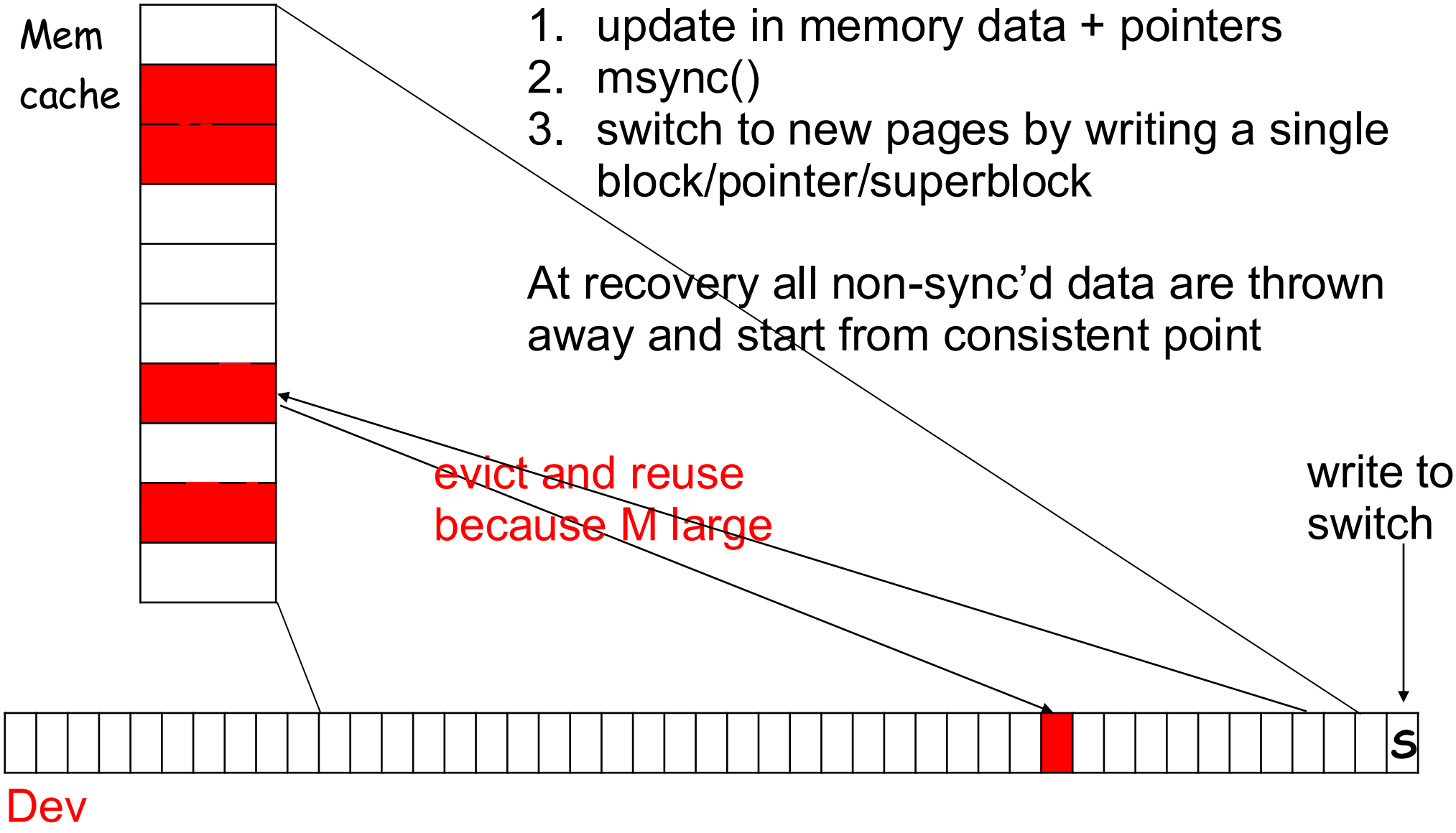
Remap-based Recovery



Remap + mmio + CoW

- Avoid syncing the two spaces on the device
- Copy-on-Write (CoW) – what is it?
  - A mechanism where a page is mapped to the process as read only
  - When the process writes the page (with a store), this causes a page fault
  - The application then allocates a new page, copies the old page to the new page
  - Perform the write to the new copy of the page, leaving old page intact (read-only)
- CoW changes how the two data copies are managed on the device
  - Consider that we have a single copy of the data on the device
  - With CoW, when a page becomes dirty, we maintain the previous version as a stable copy in memory and the device, so we should be able to recover (**how?**)
  - At the next checkpoint, flush all dirty pages as before -- msync()
  - After msync(), make new set of pages part of Dp + reclaim old pages (**how?**)
- How do we switch roles of new and old pages consistently?
  - Depends on what is being persisted – Generally:
    - Need to know all data/pages in M that point to CoW pages
    - Go to each page and modify the reference to a CoW page to point to the new page
    - These pages become themselves dirty/CoW and will be flushed at msync()
    - This continues until all references to modified data (and recursively) have been updated
  - Example
    - Consider a tree-type index – When a leaf is updated, we need to update the full path to the root so several CoW pages per data update (optimization: indirection via a table)

Remap-based Recovery



Remap + mmio + CoW

Review advantages and disadvantages

- + No logging of updates/modifications
- + Potentially, single write of data on device
- - What if M contains stuff that do not need to be persisted
  - Solved but page granularity
  - Can still be much more than necessary →
    - 1. I/O amplification, can be a lot
- - What if M does not fit in memory
  - Solved but introduced pointer updates and more CoW
  - Potentially, multiple page writes for each data update →
    - 2. No single data update anymore (WAL is 2x) -- Maybe amortize pointer writes, but unclear and depends on application
    - 3. CoW pages generate random IOs (WAL always sequential)
- - How frequent do we checkpoint?
  - Same as before – essentially, some batching interval, similar to WAL
  - Experience: at least a few seconds -- how does it compare to WAL?
- - Until checkpoint is done, we cannot write M
  - Same as before →
    - 4. Significant problem, WAL does not suffer from this
  - Consider a program where several threads update data in memory
  - At the next checkpoint, the system msync's a consistent view of data in memory
  - Since there is no record for ordering for individual updates, this requires stopping all threads that update M, until M is stable
  - **This is a global barrier, among all threads and may need to wait for more than necessary I/O due to (1)**
  - E.g. if you have 64 server threads updating M, during a checkpoint you need to block them, write to device all dirty pages, then resume
- 1,2,3,4 are significant problems -- Do general solutions exist?
  - No: (1) If updates smaller than pages, we still write full pages
  - No: (2) Unless application can amortize pointer updates, the system will do multiple writes per data update
  - No: (3) Random writes maybe ok with SSDs today and in the future, but they cost CPU
  - Maybe: (4) msync() support for concurrent updates [S. Park, T. Kelly, and K. Shen. 2013. Failure-atomic msync(): a simple and efficient mechanism for preserving the integrity of durable data. EuroSys'13. https://doi.org/10.1145/2465351.2465374]
- Remap + mmio + CoW requires careful design to make it efficient → WAL more straight-forward
  - Why?

Remap vs. WAL

- The WAL approach, implicitly, while writing the log, achieves the following
  - 1. The program selects exactly what needs to be persisted, rather than doing blind writes
    - This is a serialization process in some sense
    - Is “the program” an application or the storage system?
  - 2. The program moves updates out of working memory in **a log**
    - Threads can continue updating after log buf is written (in memory)
  - 3. The program maintains program order with **appends** to the log
    - We know exactly how to replay requests (few or many) are recreate history – simple recovery
  - 4. The WAL is written in **batches** and **sequentially** to the device
    - Avoids random writes for persisting updates, but still requires random writes for in-place
    - In KV stores there are no random writes here because of compaction
  - 5. Does not require updating other metadata on the device (5d-B)
- (1-3) are a significant benefit of WAL compared to Shadowing
  - Serialization of WAL: A burden for the system, however, with significant benefits
    - It is not transparent – the system has to record every update
    - But maintains significant information about execution
  - WAL can result in lower overhead compared to shadowing
- Issues of remap+mmio+CoW are inherent to “persistent memory” systems
  - What is ”persistent memory”?
  - Memory=Heap vs. Persistent=Recoverable
  - Heap contains a lot of stuff that need not be recovered after a failure

Reading

- Review
  - Slides + pointers, Your class notes
- Read
  - Any of the papers used as references in the slides
- Next class
  - Paper discussion