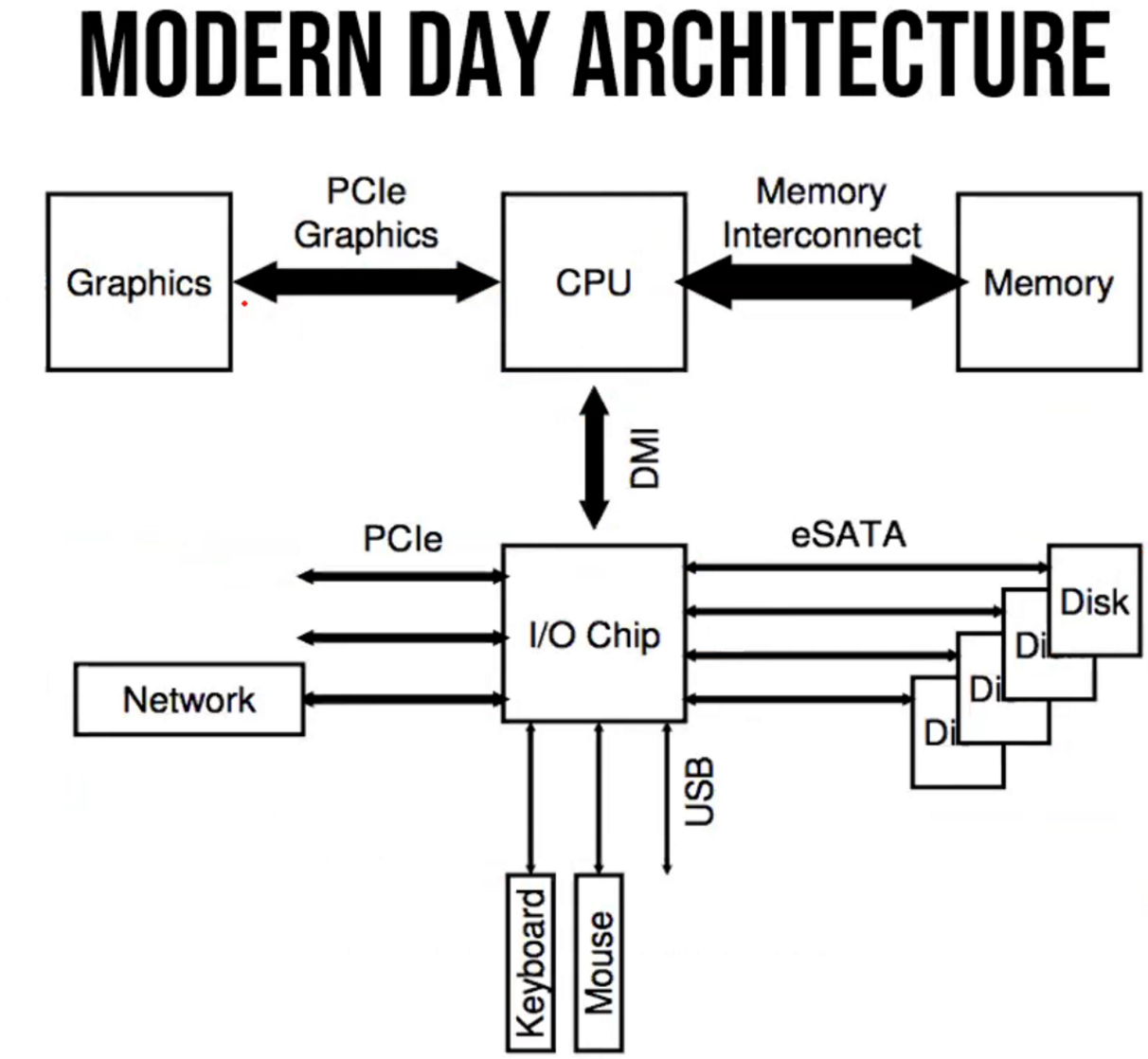
CS 537 Part 3 Study Guide By Leping Li

Sunday, August 7, 2022 9:46 PM



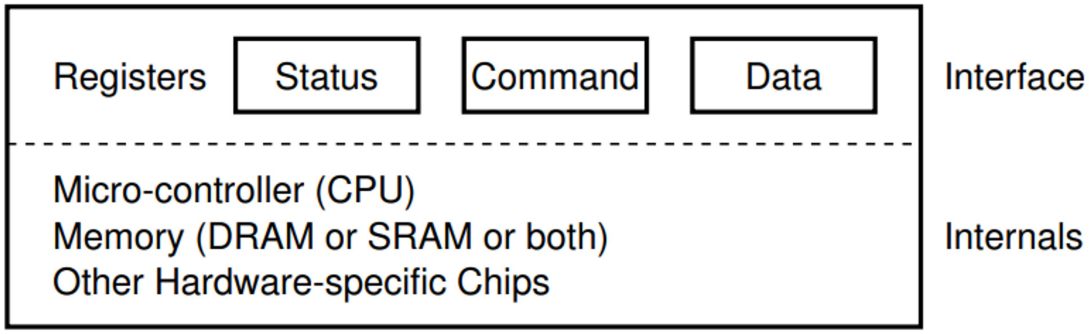


Figure 36.3: **A Canonical Device**

Interrupts vs. Polling

- Polling: the OS waits until the device is ready to receive a command by repeatedly reading the status register
- Interrupt: OS can issue a request, put the calling process to sleep, and context switch to another task.
- Fast device, better to spin than take interrupt overhead or hybrid approach
- A live lock is possible when multiple interrupts arrive at the same time.
- Can batch together several interrupts to solve live lock. A device which needs to raise an interrupt first waits for a bit before delivering the interrupt to the CPU. Cost is will increase the latency of a request

Direct Memory Access

- I/O instructions. In/Out (x86) instructions communicate with devices
- Memory Mapped I/O, hardware maps registers into memory address space. Load/Store sent to device

Hard Disk

- The drive consists of a large number of sectors (512-byte blocks), each of which can be read or written.
- Platter: a circular hard surface on which data is stored persistently by inducing magnetic changes to it. A disk may have one or more platters. Each platter has 2 sides, each of which is called surface. The platters are all bound together around the spindle, which is connected to a motor that spins the platters around (while the drive is powered on) at a constant (fixed) rate.
- 10000 Rotations Per Minute, single rotation is 6ms
- Surface is divided into rings: tracks. Stack of tracks(across platters) is called cylinder.

Reading Data from Disk

- Rotational delay time: must wait for the desired sector to rotate under the disk head
- Seek time: drive has to move the disk arm to the correct track
- Transfer time: read or write data
- Sequential: transfer dominated
- Random: seek + rotation dominated

Disk Scheduling

- the disk scheduler examines the requests and decides which one to schedule next
- Position of disk head relative to request position matters more than length of job
- First Come First Serve
  - Terrible performance with random request
- Shortest Seek Time First
  - SSTF orders the queue of I/O requests by track, picking requests on the nearest

track to complete first.
    - the drive geometry is not available to the host OS; rather, it sees an array of blocks. Solution: schedules the request with the nearest block address next.
    - Starvation
- Elevator/Scan
    - Sweep back and forth, from one end of disk to other
    - Serving requests as the arm passes that cylinder
    - Sorts by cylinder number, ignore rotation delays
    - C-Scan: only sweep in one direction, more fair to inner and outer tracks.
- Work conserving schedulers always do work if work exists
- Non-work conserving schedulers wait if anticipate another request will arrive

Redundant Array of Inexpensive Disks(RAID)

- a technique to use multiple disks in concert to build a faster, bigger, and more reliable disk system.
- Consistent-update problem: ensuring different replicas have the same data if a crash occurs.
    - Solution: journaling, have a journal/log of what you do
- To evaluate a RAID
    - Capacity: how much useful capacity is available to clients of the RAID?
    - Reliability: How many disk faults can the given design tolerate?
    - Performance
- RAID-0: Striping
    - No redundancy
    - stripe blocks across the disks of the system
    - Blocks in the same row are called a stripe
    - Arrange the blocks across disks in chunk, large chunk size reduces positioning time but also reduce parallelism
    - Ensures maximum performance and capacity, but at cost of reliability

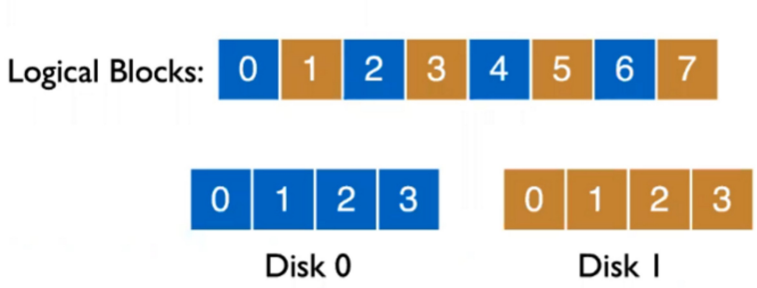
Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 38.1: **RAID-0: Simple Striping**

-

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 38.1: **RAID-0: Simple Striping**

- 

-

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size:
1	3	5	7	2 blocks
8	10	12	14	
9	11	13	15	

Figure 38.2: **Striping With A Bigger Chunk Size**

-

What is capacity?	**N * C**
How many disks can fail (no loss)?	**0**
Latency?	**D**
Throughput (sequential, random)?	**N*S , N*R**

- RAID-10: Mirroring + Striping
  - Stripes of mirroring

- 2 disks

Disk 0	Disk 1
0	0
1	1
2	2
3	3

- 4 disks

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

- RAID-1: Mirroring
  - make more than one copy of each block in the system
  - Always handle 1 disk failures, may handle N/2 if to different replicas

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: **Simple RAID-1: Mirroring**

What is capacity?	**N/2 * C**
How many disks can fail?	**1 (or maybe N / 2)**
Latency (read, write)?	**D**

What is steady-state throughput for	
- random reads?	**N * R**
- random writes?	**N/2 * R**
- sequential writes?	**N/2 * S**
- sequential reads?	**Book: N/2 * S (other models: N * S)**

- RAID-4: Parity
  - Have a parity block that stores the redundant information for that stripe of blocks.
  - To compute parity, use XOR

- Number of 1s in any row, including the parity bit, must be an even number
- Write out new parity: P_new = (C_old XOR C_new) XOR P_old
- Small-write problem, the parity disk prevents any parallelism from materializing; all writes to the system will be serialized because of the parity disk

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.5: **Full-stripe Writes In RAID-4**

What is capacity?	(N-1) * C
How many disks can fail?	1
Latency (read, write)?	D, 2*D (read and write parity disk)

how to avoid parity bottler

What is steady-state throughput for	
- sequential reads? (N-1)S	(N-1) * S
- sequential writes? (N-1)S	(N-1) * S (parity calculated for full stripe)
- random reads? (N-1)xR	(N-1) * R
- random writes? R	R/2 (read and write parity disk)

- RAID-5: Rotating Parity
  - Rotates the parity block across drives
  - Strictly better than RAID-4

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figure 38.7: **RAID-5 With Rotated Parity**

What is capacity?	(N-1) * C
How many disks can fail?	1
Latency (read, write)?	D, 2*D (read and write parity disk)

What is steady-state throughput for RAID-5?

- sequential reads? (N-1) * S
- sequential writes? (N-1) * S
- random reads? (N) * R
- random writes? N * R/4

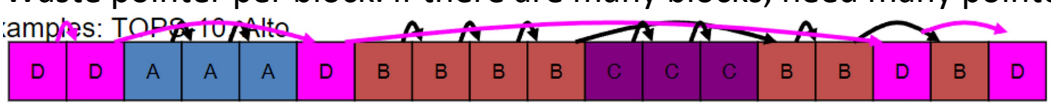
File System

- Collection of files
- File has three types of names
  - Unique id: inode numbers
    - Inodes are unique within file system images
    - Different file system may use the same number
    - Numbers may be recycled after deletes
    - Stored in known, fixed block location on disk
  - Path
    - String name, path to inode mapping
    - Traverse directory tree is expensive
  - File descriptor
    - Do expensive traversal once
    - Store inode in descriptor object(in memory)
    - Do read/write via descriptor, which tracks offset
    - Each process has a file descriptor table
  - Delete files
    - Do not delete data, just remove the file name - inode mapping
    - Inode is freed when there are no references
- Link
  - Increment reference count in inode whenever add link
  - Cannot hard link directories because that may cause a cycle
  - Soft links point to second path name, will have new inode number, do not increment reference count

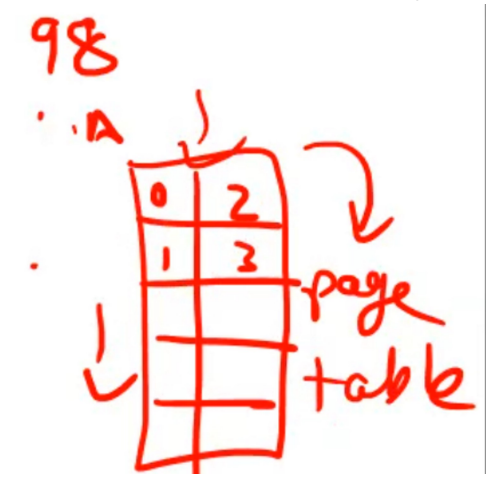
- File system keeps newly written data in memory for a while(buffer cache), useful for reads, writes. Fsync(int fd) forces buffer to flush from memory to disk
- Rename, deletes an old link to a file, create a new link to a file, has to be atomic
- Contiguous Allocation
  - Allocate each file to contiguous sectors on disk
  - Meta-data: starting block and size of file
    - Must predict future file size
  - OS allocates by finding sufficient free space
  - Horrible external fragmentation(need periodic compaction)
  - May not be able to grow without moving
  - Excellent performance for sequential access
  - Simple calculation for random access(you have the starting block and file size)
  - Small overhead for meta-data
  - 

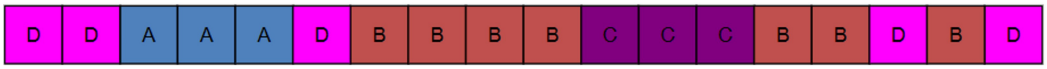
- Small # of Extents
  - Allocate multiple contiguous regions(extents) per file
  - Similar to segmentation technique
  - Meta-data: small array(2 - 6) designating each extent, starting block and size for each entry
  - Helps external fragmentation until out of extents
  - Can grow but may run out of extents
  - Good performance for sequential access
  - Simple calculation for random access
  - Small overhead for meta-data
  - 

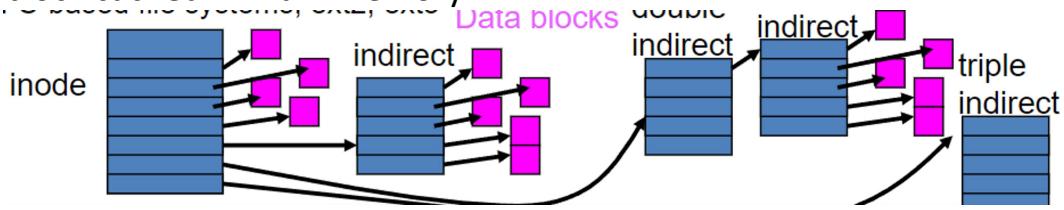
  - 

- Linked Allocation
  - Allocate linked-list of fixed-size blocks(multiple sectors)
  - Meta-data: location of first block of file, each block also contains pointer to next block
  - No external fragmentation, internal possible
  - Can grow easily
  - Performance of sequential access depends on data layout. Potential many seeks
  - Ridiculously poor performance for random access, has to traverse blocks
  - Waste pointer per block. if there are many blocks, need many pointers
  - 

- File-Allocation Table
  - Variation of linked allocation, keep linked-list information for all files in on-disk FAT

table
  - Meta-data: location of first block of file, and FAT table
  - Read from two disk locations for every data read
  - Cache FAT in main memory
  - 

- Indexed Allocation
  - Allocate fixed-size blocks for each file
  - Meta-data: fixed-size array of block pointers
  - Allocate space for pointers at file creation time
  - No external fragmentation
  - Files can be easily grown to max file size
  - Support random access
  - Large overhead for meta-data
  - 

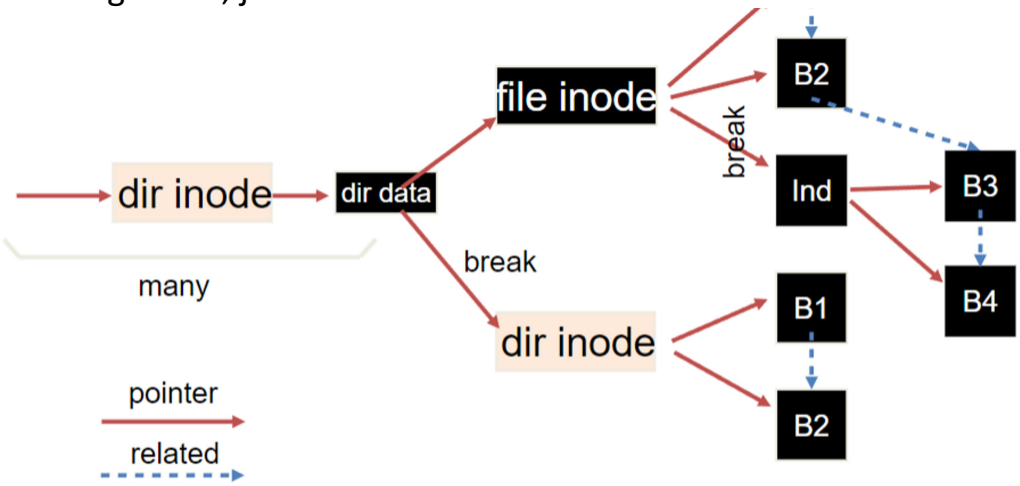
- Multi-Level Indexing
  - Variation of indexed allocation
  - Dynamically allocate hierarchy of pointers to blocks as needed
  - Meta-data: small number of pointers allocated statically, additional pointers to blocks of pointers
  - Do not waste space for unneeded pointers
  - Fast access to small files
  - Can grow to very large size
  - Need to read indirect block of pointers to find address(extra read), can keep indirect block cached in main memory
  - 

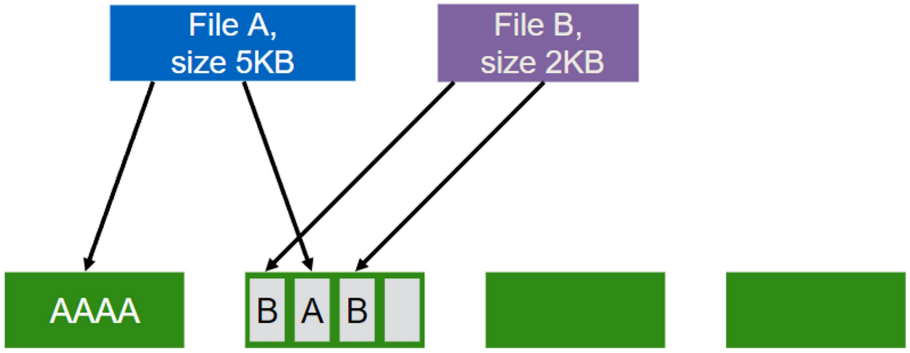
Old File System

- Free list are embedded in inodes, data blocks.

- Achieved 2% of potential performance
- Over time, files are created and deleted. File system is fragmented. Free list makes contiguous chunks hard to find
- Occasionally defragment disk, move around existing files, keep free list sorted
- Block size affects performance, large block size require less rotations and seeks. Also fewer indirect blocks. However, OFS has small blocks.
- Distance between inode and data can affect performance, OFS blocks laid out poorly
- Treats disk like RAM

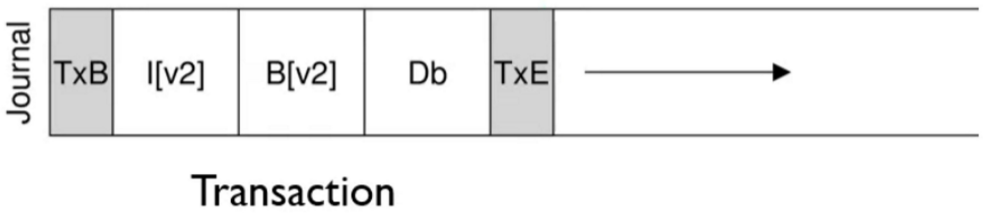
Fast File System

- Use bitmap instead of free list. Bitmaps provides better speed, with more global view(load things in memory)
- Fast to update, just flip single bit to change state
- Fast to lookup, reading one bitmap block gives state of many data blocks
- Keep data near its inode, inodes in same directory should be near one another
- Groups were ranges of cylinders
- Allocate file inodes in the same group with dir.
- Allocate dir inodes in a new group with fewer used inodes than average group
- Split large files into many small chunks, starting at indirect, put blocks in a new block group. Each chunk corresponds to one indirect block
- Large block improve performance but waste space(most files are small)
- Introduce fragment for files that use parts of blocks, only tail of file uses fragments. If lots of fragments, just make a new block
- 

- 

File System Consistency

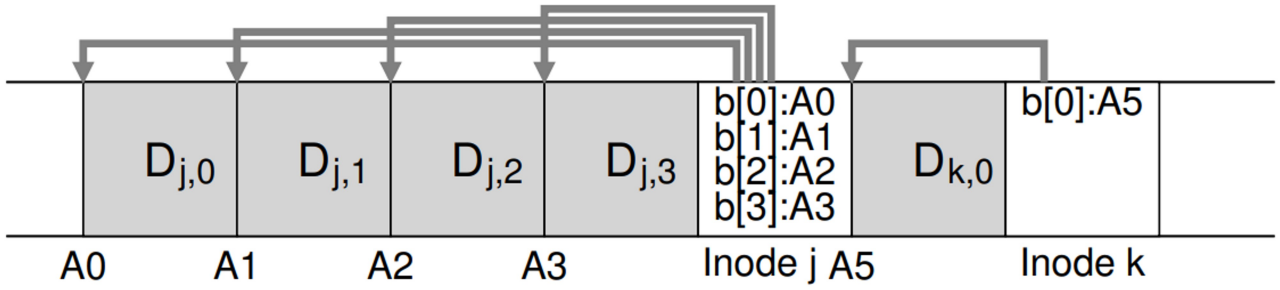
- If file system is interrupted between writes, may leave data in inconsistent state
- Only single sector writes are guaranteed to be atomic by disk
- Crash Scenario
  - Just the data block is written to disk, but no inode points to it, no bitmap says the block is allocated. Data would be lost, but consistency.
  - Just the updated inode is written to disk, data block is not written, may read garbage data, and inode points to a block that is not allocated in bitmap, inconsistency.
  - Just the updated bitmap is written to disk, bitmap indicates a block is allocated but there is no inode points to it. Space leak, inconsistency.
  - Inode, bitmap are written to disk but not data. May read garbage data but consistency
  - Inode and data are written to disk, but not bitmap. Inode points to the correct data on disk but there is an inconsistency between inode and bitmap. Data block may get overwritten.
  - Bitmap and data are written to disk, but not inode. There is an inconsistency between inode and bitmap. Space leak since a data block is allocated but no inode points to it.
- File System Checker(FSCK)
  - After crash, scan whole disk for contradictions and fix if needed
  - Keep file system offline until repaired
  - Can keep the file system to a consistent state but not correct state(do not know what is the correct state)
  - Very slow
  - Check superblocks, make sure that file system size is greater than number of blocks have been allocated
  - Read every valid inode + indirect block to fix bitmap
  - Check inode state, make sure it has a valid file type
  - Verfies inode links, scan through the entire directory tree, build its own link count

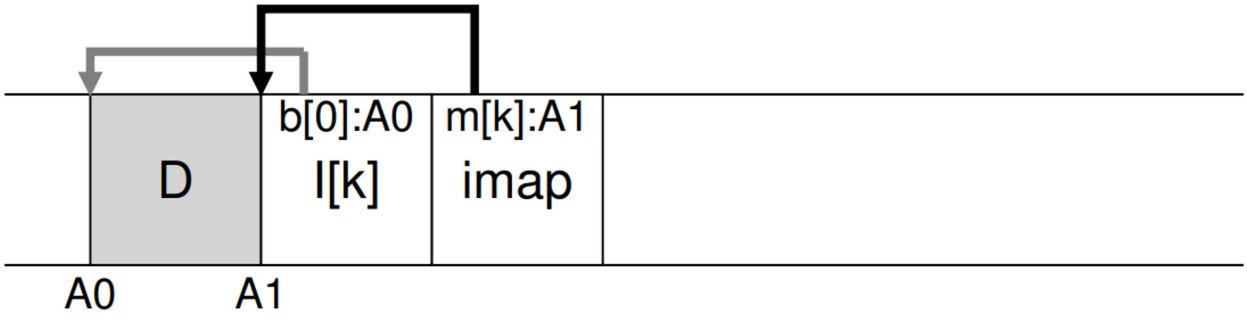
then compare
    - If an allocated inode is discovered but no directory refer to it(link count == 0), move it to lost-found directory
    - Check for duplicate pointers(two inode points to the same block). If one inode is obviously bad, it may be cleared.
    - If an inode points to something outside its valid range, remove the pointer from the innode or indirect block.
    - Check directories, make sure "."(root directory) and ".."(itself) are the first entries and each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy.
- Journaling
    - Do not delete any old info until all new info is safely on disk
    - Make a note of what needs to be written
    - After note is completely written, can update metadata and data
    - Remove note
    - If crash, if note is not completely written, ignore note(old data still good), if not is completely written, just replay to get new data
    - 

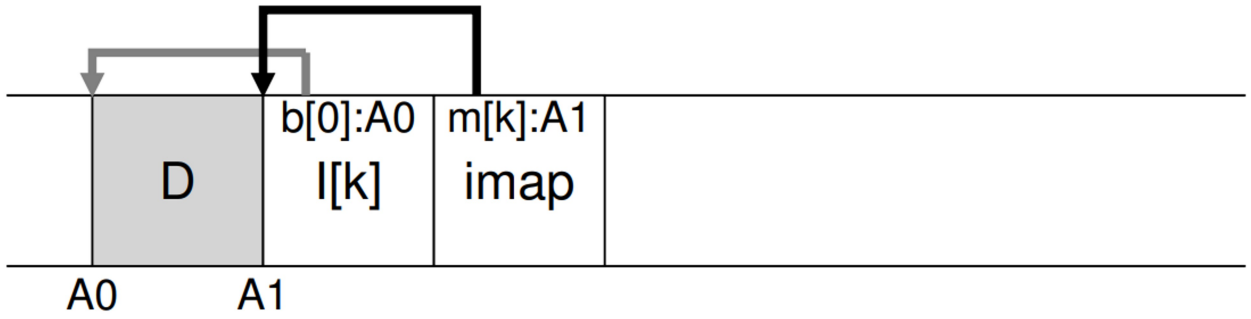
Transaction
    - TxB: Journal descriptor block, describes contents of transactions
    - TxE: Journal commit block
    - Batch transactions to improve performance
    - Need to journal transactions entries complete before journal commit
    - Need to ensure journal commit complete before checkpoint
    - Need to ensure checkpoint complete before free journal
    - Checksum optimization
        - Reduces strict ordering
        - In last transaction block, store checksum of rest of transaction, so there is no barrier between last journal transaction entry and commit block
    - Write Buffering optimization
        - Batch many updates into one transaction
        - Delay checkpoints, since checkpointing is random, if do lots of checkpointing together, might be sequential
    - Metadata Journaling
        - Do not write data block to journal, only write metadata
        - Write data blocks to the disk first before related metadata is written to disk
        - Writeback mode: do not control ordering of data + metadata

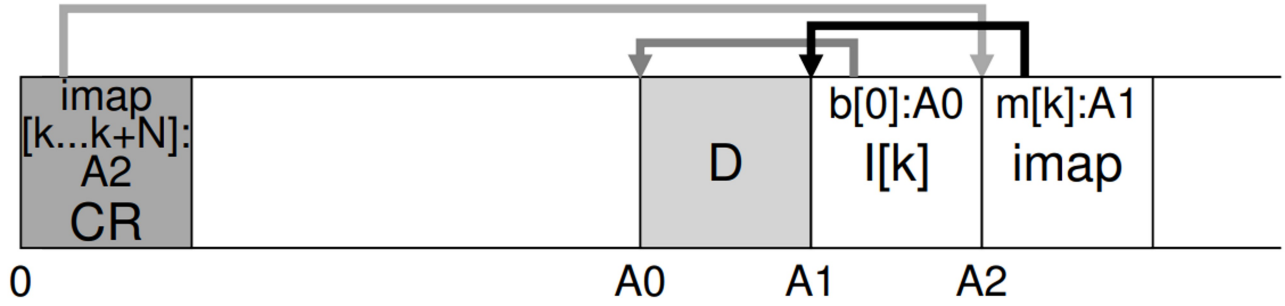
- Logical Journaling
  - Record changes to bytes, not contents of block
  - For recovery, need to read existing contents of in-place data and reapply changes

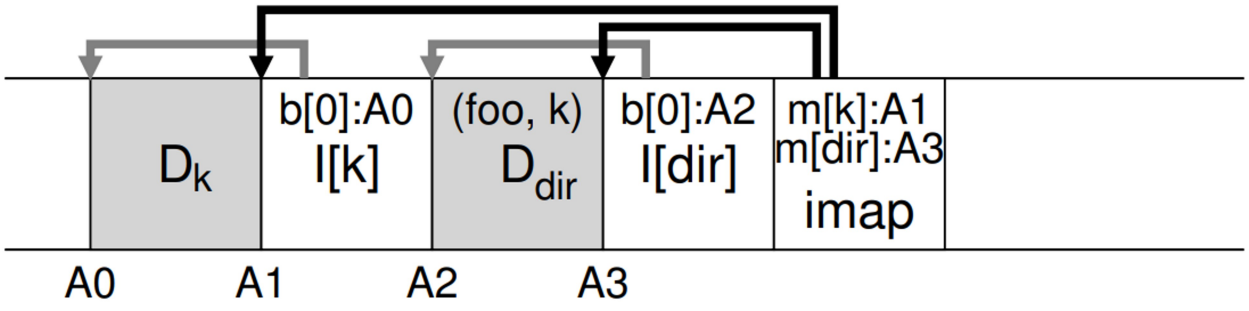
Log-structured File System

- As memory gets bigger, more data can be cached in memory, disk traffic increasingly consists of writes, as reads are serviced by cache. The performance of file system is determined by writes
- There is a large gap between random I/O performance and sequential I/O performance. If you are able to use disks in a sequential manner, you gain a sizeable performance advantage over approaches that cause seeks and rotations.
- FFS would perform a large number of writes to create a new file of size one block: new inode, update inode bitmap, directory data block, update directory inode, new data block, update data bitmap….. Too many seeks and rotations
- File systems do not avoid worst-case RAID writing behavior(small-write problem)
- When writing to disk, LFS first buffers all updates (including metadata!) in an in memory segment; when the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk. LFS never overwrites existing data, but rather always writes segments to free locations. Because segments are large, the disk (or RAID) is used efficiently, and performance of the file system approaches its zenith.
- 

- Finding inodes is difficult since inodes are throughout the disk.
  - Inode map is introduced. The imap is a structure that takes an inode number as input and produces the disk address of the most recent version of the inode.
  - when appending a data block to a file k, LFS actually writes the new data block, its inode, and a piece of the inode map all together onto the disk
  - 

- 

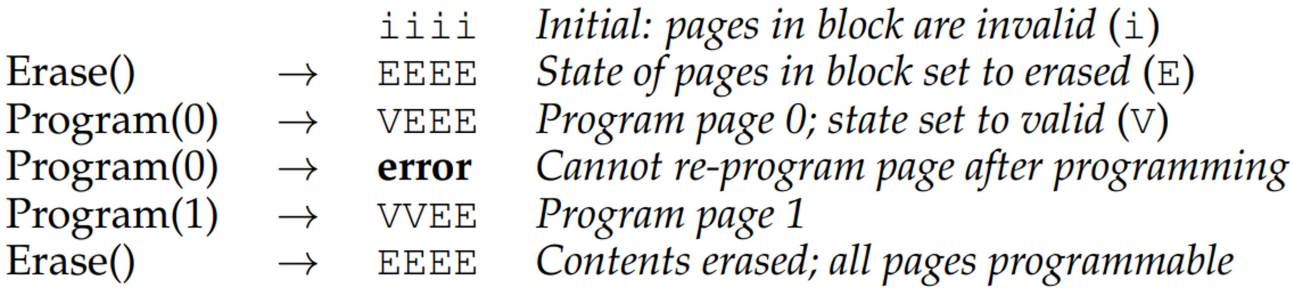
  - Checkpoint region contains pointers to the last pieces of the inode map and it is at a fixed place on disk. It is updated periodically.
- 

- To read a file
  - The first on-disk data structure we must read is the checkpoint region. The checkpoint region contains pointers (i.e., disk addresses) to the entire inode map, and thus LFS then reads in the entire inode map and caches it in memory. After this point, when given an inode number of a file, LFS simply looks up the inode-number to inode-diskaddress mapping in the imap, and reads in the most recent version of the inode. To read a block from the file, at this point, LFS proceeds exactly as a typical UNIX file system, by using direct pointers or indirect pointers or doubly-indirect pointers as need be.
- Directory structure is basically identical to classic UNIX file systems.
  - 

- Garbage Collection
  - Repeatedly writes the latest version of a file to new locations on disk
  - Leaves old versions of file structures scattered throughout the disk
  - Needs to clean old versions of files
  - LFS clean up large chunks of space for subsequent writing. Periodically, the LFS cleaner reads in a number of old (partially-used) segments, determines which blocks

are live within these segments, and then write out a new set of segments with just the live blocks within them, freeing up the old ones for writing.
Specifically, we expect the cleaner to read in M existing segments, compact their contents into N new segments (where N < M), and then write the N segments to disk in new locations. The old M segments are then freed and can be used by the file system for subsequent writes.
    - LFS must be able to determine live files. Each segment has a summary block.
- Crash and Recovery
    - When updating the checkpoint region. LFS first writes out a header, the body of CR, then finally one last block. If a crash happens, LFS can detect this by seeing an inconsistent pair of timestamps.
    - LFS starts with the last checkpoint region, read backward. Even though CR gets updated, it still can recover much of the data and metadata written since the last checkpoint.

Flash-based SSDs

- Flash chips are designed to store one or more bits in a single transistor; the level of charge trapped within the transistor is mapped to a binary value.
- Flash chips are organized into banks or planes which consist of a large number of cells.
- A bank is accessed in two different sized units: blocks (sometimes called erase blocks), which are typically of size 128 KB or 256 KB, and pages, which are a few KB in size (e.g., 4KB). Within each bank there are a large number of blocks; within each block, there are a large number of pages.
- A flash chip supports three operations: read, erase, and program. Last two are used to write
    - Read(a page): can read any page, very fast, regardless of location on the device(random access)
    - Erase(a block): before writing to a page, needs to erase the entire block, setting bits to 1, quite expensive. Need to copy any data that we care about before erase
    - Program(a page): write contents by clearing some bits, less expensive than erasing but more costly than reading
    - 

Device	Read (μs)	Program (μs)	Erase (μs)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Figure 44.2: **Raw Flash Performance Characteristics**

- when a flash block is erased and programmed, it slowly accrues a little bit of extra charge. Over time, as that extra charge builds up, it becomes increasingly difficult to differentiate between a 0 and a 1. At the point where it becomes impossible, the block becomes unusable.
- When accessing a particular page within a flash, it is possible that some bits get flipped in neighboring pages; such bit flips are known as read disturbs or program disturbs, depending on whether the page is being read or programmed, respectively.
- The flash translation layer, or FTL takes read and write requests on logical blocks (that comprise the device interface) and turns them into low-level read, erase, and program commands on the underlying physical blocks and physical pages (that comprise the actual flash device).
  - Direct mapped FTL
    - A read to logical page N is mapped directly to a read of physical page N
    - For a write to logical page N, the FTL first reads in the entire block that page N is, then erase the block, then programs.
    - Terrible performance and reliability
  - A Log-structured FTL
    - Upon a write to logical block N, the device appends the write to the next free spot in the currently-being-written-to block; we call this style of writing logging. To allow for subsequent reads of block N, the device keeps a mapping table (in its memory, and persistent, in some form, on the device); this table stores the physical address of each logical block in the system.
    - In-memory mapping table for read
    - Garbage is created because of log-structured, has to do garbage collection
      - find a block that contains one or more garbage pages, read in the live (non-garbage) pages from that block, write out those live pages to the log, and (finally) reclaim the entire block for use in writing.
      - To check live data, read map
      - Cleaning can be delayed by adding extra flash capacity
    - Mapping Table
      - Size can be very large
      - Instead of per page, only keep a pointer per block, performance is bad due to small write.

        - Hybrid mapping, keeps a few blocks erased and directs all writes to them. Keep per-page mappings for these log blocks. Keep the number of log blocks small. If overwrite happens, old block becomes log block
    - Page Mapping with Caching
        - Cache only the active parts of the FTL in memory
- Wear Leveling
    - Spread data across all the blocks of the device evenly to prevent wearing out.
    - The basic log-structuring approach does a good initial job of spreading out write load, and garbage collection helps as well.
    - sometimes a block will be filled with long-lived data that does not get over-written; in this case, garbage collection will never reclaim the block, and thus it does not receive its fair share of the write load. To remedy this problem, the FTL must periodically read all the live data out of such blocks and re-write it elsewhere, thus making the block available for writing again.