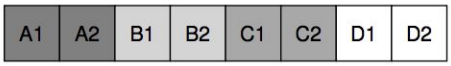
Chapter 41: FFS Implementation

41.1: The Problem: Poor Performance

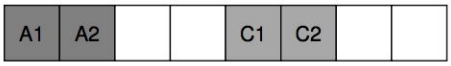
- The old, simple file system that was created by Ken Thompson for Unix, was extremely simple
    - had a superblock, a list of inodes, and data blocks
    - the superblock contained info on how big the volume was, how many inodes there are, and a pointer to the head of a free list of blocks
- This implementation has terrible performance + it gets worse with time
- The main issue was that the old UNIX file system treated the disk like it was a random-access memory -> meaning data was spread all over the place without regards to how expensive positioning is
    - The data blocks were often very far from its inode, inducing an expensive seek whenever one first read the inodes
- Must be careful with data blocks because misuse can lead to data fragmentation

the disk, thus reducing performance dramatically.

For example, imagine the following data block region, which contains four files (A, B, C, and D), each of size 2 blocks:



If B and D are deleted, the resulting layout is:



As you can see, the free space is fragmented into two chunks of two blocks, instead of one nice contiguous chunk of four. Let's say you now wish to allocate a file E, of size four blocks:

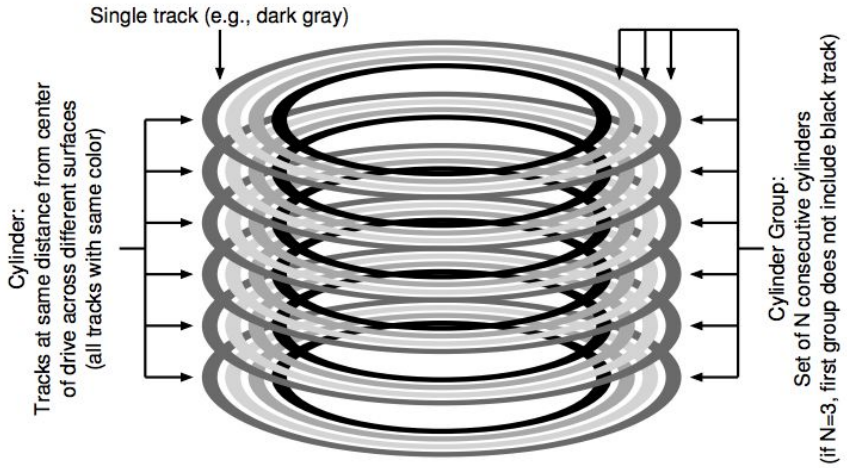
- This means that E is spread across the disk and therefore additional seek time is needed and performance is reduced due to fragmentation
    - Disk defragmentation aims to fix this problem
- Therefore, we must think, how can we organize file system data structures so as to improve performance? What types of allocation policies do we need on top of those data structures? -> must make file system disk aware

41.2: FF: Disk Awareness is the Solution

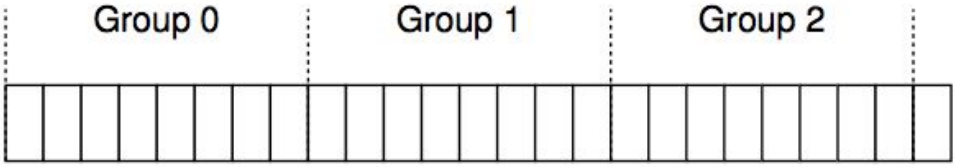
- A group at Berkeley decided to create the FFS (Fast File System) which is disk aware
    - kept the same interface to the file system (the same APIS, including open(), read(), write, lseek(), etc.) but changed the internal implementation

41.3: Organizing Structure: The Cylinder Group

- FFS divides the disk into a number of **cylinder groups**
- A single cylinder is a set of tracks on different surfaces of a hard drive that are the same distance from the center of the drive



- FFS aggregates each N consecutive cylinders into group, and thus the entire disk can be viewed as a collection of cylinder groups
- Modern file systems instead organize the disk drive into block groups, each of which is just a consecutive portion of the disk's address space



- By splitting the blocks into cylinder groups/block groups, placing files in the same group can ensure that accessing one after the other will not result in long seeks across the disk
- FFS must have the ability to place and store files/directories into these groups

- FFS keeps the above diagram in a single cylinder group in order to place files in certain cylinder groups
    - FFS keeps a single copy of the superblock in each group for reliability reasons
        - needed to mount the system so that if one copy becomes corrupt, you can still mount and access the file system by using a working replica
    - There is also a per-group inode bitmap and data bitmap to keep track of which inodes/data blocks are allocated (free/used)
        - we can avoid some of the fragmentation problems because bitmaps can be used to easily find a large chunk of free space and allocate it to a file

41.4: Policies: How to Allocate Files and Directories

- The basic mantra for deciding how to place files and directories (metadata) on disk to improve performance: **keep related stuff together**
- **FFS must decide what is related and place it within the same block group**
- Heuristics for the placement of directories
    - Find the group with the lowest number of allocated directories, and a high number of free inodes, and put the directory data and inode in that group
- Heuristics for files
    - First, it makes sure to allocate the data blocks of a file in the same group as the inode -> prevent long seeks from occuring
    - Second, it places all files that are in the same directory in the cylinder group of the directory they are in
    - The data blocks of each file are near each file's inode and files in the same directory are near one another

41.5: Measuring File Locality

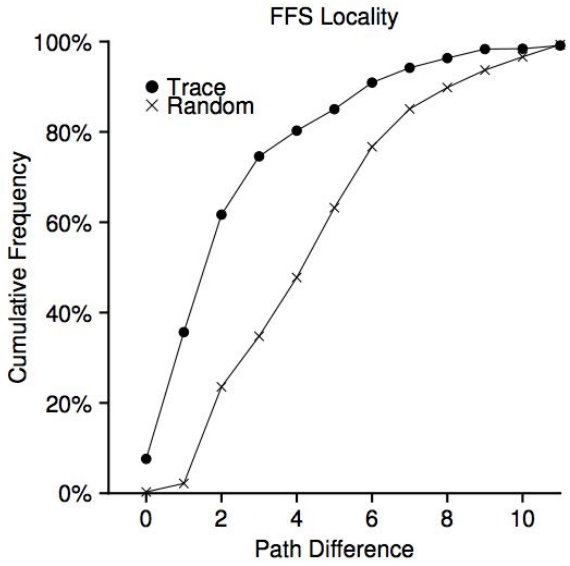


Figure 41.1: **FFS Locality For SEER Traces**

-
- Using the SEER traces, we'll analyze how "far away" file accesses were from one another in the directory tree
- Ex: if file f is opened, and then re-opened next in the trace (before any other files are opened), the distance between those two opens in the directory is zero
- If f opened, then g opened, the distance between the access is 1
- Distance metric measures how far up the directory tree you have to travel to find the common ancestor of the two files -> the closer they are in the tree, the lower the metric

41.6: The Large-File Exception

- In FFS, there is one important exception to the general policy of file placement, and it arises for large files
    - A big file could possibly use up all of the blocks in the block group, allowing other files to not exhibit the locality performance improvements
- For Large files, FFS does the following
    - After some number of blocks are allocated into the first block group, FFS places the next “large” chunk (e.g. those pointed to by the first indirect block) of the file in another block group. Then, the next chunk of the file is placed in yet another different block group, and so on
- **Amortization:** Process of reducing an overhead by doing more work per overhead paid
    - By giving a large chunk size, the file system will spend most of its time transferring data from disk and just a relatively little time seeking between chunks of the block

41.7: A Few Other Things about Amortization

- Accommodating small files
    - Small files with large blocks could cause internal fragmentation
- Introduced sub-blocks
    - 512 byte little blocks that the file system could allocate to files
    - A small file could use several sub blocks instead of an entire block -> no fragmentation
    - Allocate 512-byte blocks until it fills up the full 4KB of data -> copy the sub-blocks into another 4KB data block and then
- Process could be inefficient, requiring a lot of extra work for the file system
    - extra I/O to perform the copy
    - Avoid this through buffering writes by modifying the libc library -> and then issue the 4KB chunks to the file system
- Problem arose with FFS (file placed on consecutive sectors of the disk)
    - during sequential reads, FFS would first issue a read to block 0
        - by the time it was complete, the FFS issued a read to block 1 -> block rotated under the head and now the read to block 1 would now have to incur a full rotation
    - Fixed this problem through skipping every other block -> parametrization
- Modern disks are smarter
    - Internally read the entire track in and buffer in in an internal disk cache (called the **track buffer)**
    - Subsequent reads to the track will just return the desired cache data

Chapter 42: Crash Consistency: FCSK and Journaling

- The file system manages a set of data structures to implement the expected abstractions: files, directories, and all of the other metadata needed to support the basic abstraction that we expect from a file system

- Major challenge: How to update persistent data structures despite the presence of a power loss or system crash
- We'll begin by examining the approach taken by older file systems, known as fsck or the **file system checker**

42.1: A Detailed Example

- Journaling: Also known as write-ahead logging: adds a little bit of an overhead to each write but recovers more quickly when a crash occurs
- **Workload** that updates on-disk structures in some way
    - assume simple workload: the append of a single data block to an existing file
- When we append a file, we are adding a new data block to it, and thus must update three on disk structures
    - inode, the new data block, and a new version of the data bitmap
    - thus, we have three blocks which we must write to disk
- Writes to disk do not occur immediately
    - Rather, they will sit in main memory (in the page cache/buffer cache) for some time -> when the file system finally decides to write them to disk -> issues request to disk
- **Crash Scenarios**
    - **Just the data block is written to disk:** the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated -> same as if the write never occured -> not a problem for crash consistency
    - **Just the updated inode is written to disk:** the inode points to the disk address where the data is going to be written to but it has not. Thus if we trust the pointer, **we can read garbage data from the disk**
        - Disagreement between the inode and bitmap (inode says it has been allocated but bitmap does not) is called **file-system inconsistency**
    - **Just the updated bitmap is written to disk:** the bitmap indicates that the block is allocated, but there is no inode that points to it -> the block could never be used in the file -> **space leak**
    - **The inode and bitmap are written to disk, but not data:** The file system metadata is completely consistent: the inode has a pointer to the block, the bitmap indicates that the block is in use, and thus everything looks OK from the perspective of the file system's metadata -> however, the data has garbage values
    - **The inode and the data block are written, but not the bitmap:** we have the inode pointing to the correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap
    - **The bitmap and data block are written, but not the inode**: Inconsistency between the inode and the bitmap. Even though the data block and bitmap are written, we have no idea which file it belongs to because there is no inode to point to the file
- **The Crash Consistency Problem**

    - What we'd like to do is to move the system from one consistent state to another atomically
    - Difficult to do because the disk only commits one write at a time, and crashes may occur during disk writes

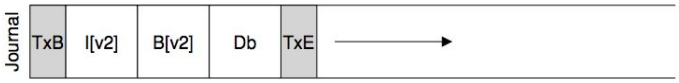
42.2: Solution #1: The File System Checker

- Let inconsistencies happen until we have to reboot -> lazy approach
- FCSK is run before the file system is mounted and made available -> once finished, the on-disk file system should be consistent and thus accessible to users
- **Superblock:** FCSK checks if the superblock looks reasonable, mostly doing sanity checks such as making sure the file system size is greater than the number of blocks that have been allocated
- **Free blocks:** Next, fcsk scans the inodes, indirect blocks, double indirect blocks, etc, to build an understanding of which blocks are currently allocated within the file system
    - Uses the knowledge to produce correct version of the allocation bitmaps -> resolves inconsistency
- **Inode State:** Each inode is checked for corruption or other problems. Ex: fcsk checks whether each allocated inode has a valid type field (e.g. regular file, directory, symbolic link, etc.) If there are problems, the inode is considered suspect and is cleared by fcsk and the bitmap is updated
- **Inode links:** fcsk also verifies the link count of each allocated inode
- **Duplicates:** Checks for duplicate pointers. I.e, cases where two different inodes refer to the same block. If one inode is obviously bad, it may be cleared.
- **Bad Blocks:** A check for bad block pointers is also performed while scanning through the list of all pointers.
    - Considered "bad" if it obviously points to something outside its valid range. E.g. address that refers to a block greater than the partition size. In this case, the fcsk just removes/clears the pointer from inode/indirect block
- **Directory checks:** fcsk does not understand the contents of user files
    - However, directories hold specifically formatted information created by the file system itself. Thus, fcsk performs additional integrity checks on each directory content
    - Makes sure that . and .. are the first entries and that each inode referred to in the directory entry is allocated.
    - Ensures that no directory is link to more than once in the hierarchy
- Performance of fcsk are largely dependent on the file system and can be extremely slow.

42.3: Solution #2: Journaling (Write Ahead Logging)

- First file system to use journaling was Cedar
- Basic idea
    - When updating the disk, before overwriting the structures in place, first write down a little note (somewhere else on disk, in a well-known location) describing what you are about to do

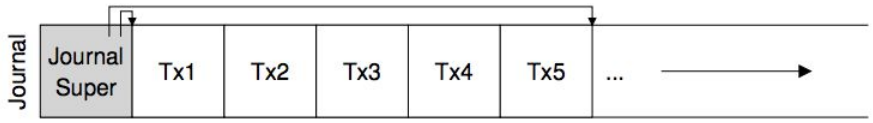
- By writing a note to disk, you are guaranteeing that if a crash takes place during the overwrite of the structures, you can go back and look at the note to try again
- **Linux et3: A popular journaling file system**
    - Disk is divided into block groups where each block has an inode and bitmap as well as inodes and data blocks
    - New structure is the journal itself, which takes up a small amount of space next to the superblock
- **Data Journaling**



    -
    - **Includes information about the pending update to the file system (final addresses of the blocks), transaction to tell us about update, and some kind of transaction identifier (TID)**
    - The middle three blocks contain the exact contents of the blocks themselves
        - **physical logging:** we put the exact physical contents of the update in the journal
        - **logical logging:** puts a more compact logical representation of the update in journal (can save space but is more complex)
    - **Checkpointing:** Once the transaction is safely on disk, we can overwrite the old structures in the file system
        - Issue the writes to their disk locations
    - Sequence of operations (Summary)
        - Journal Write: write the transaction, including the transaction begin/end blocks including the TID and the pending data to be written
        - Checkpoint: write the pending metadata and data updates to their file system locations
    - A problem can occur when a crash occurs during a transaction/write to the journal
        - If we write the blocks individually, this would be safe but will significantly slow down performance
        - Sequential writes are much faster but can be unsafe
            - Unsafe because the disk can internally perform scheduling and complete small pieces of the big write in any order
            - Could copy contents of a garbage block that is not written to the correct location
            - Could be critical if this happens to something like a superblock
    - Solution: file system issues the write in two steps
        - Writes all blocks except the transaction end block to the journal, issuing these writes at once. (Meaning that the journal will not include an end block)
        - When the writes complete, the file system issues the write of the transaction end block and leaves the journal in its final, safe state

    - **Important Aspect: Atomicity guarantee provided by the disk**
        - DIsk may guarantee that any 512-byte write will either happen or not (and never be written)
        - We must make sure that the write to the transaction end block is atomic so therefore it should take a single 512-byte block
    - Final Steps
        - Journal Write: Write the contents of the transaction to the log (except for transaction end block) and wait for writes to complete
        - Journal Commit: write the transaction commit block (containing the end point) to the log, and wait for completion
        - Checkpoint: Write the contents of the update to their final disk locations
- **Recovery**
    - If a crash occurs before the transaction is written safely to the log, the pending update is simply skipped
    - If a crash occurs after the transaction has been committed to the log but before the checkpoint the file system can recover the update
        - Transactions are **replayed** as the file system recovery process scans the log and looks for transactions that have committed to the disk. -> **redo logging**
    - if is fine for a crash to happen at any point during checkpoint, even after some of the updates to the final locations of the blocks have completed. In the worst case, some updates are simply performed again
- **Batching Log Updates**
    - Basic protocol could add a lot of extra disk traffic. Ex: creating two files in a row in the same directory
    - Some file systems do not commit each update to disk one at a time
    - rather, one can buffer all updates into a global transaction. In our example above, when the two files are created, the file system just marks the in-memory inode bitmap, inodes of the files, directory data, and directory inode as dirty -> adds them to the list of blocks
    - Single global transaction is committed containing all the updates
    - **Main goal: to avoid excessive write traffic**
- **Making the Log Finite**
    - We thus have arrived at a basic protocol for updating file-system on-disk structures. The file system buffers updates in memory for some time; When it is finally ready to write to disk, the file system first carefully writes out the details of the transaction to the journal (aka write-ahead log)
    - **Problems when the log becomes full**
        - the larger the log, the longer recovery will take as the recovery process must replay all the transactions within the log
        - When the log is full/nearly full, no further transactions can be made and the file system essentially becomes useless
    - Addressing these problems

        - **treat the log as a circular data structure and reuse it over and over**
            - **sometimes referred to as a circular log**
        - To do so, the file system must take action some time after a checkpoint
        - Specifically, once a transaction has been checkpointed, the file system should free the space it was occupying within the journal
        - You can mark the oldest and newest non-checkpointed transactions in the log in a **journal superblock** and all other space is free



        -
        - In the journal superblock, the journaling system records enough information to know which transactions have not yet been checkpointed
        - **Journal write:** Write the contents of the transaction (containing TxB and the contents of the update) to the log; wait for these writes to complete
        - **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction is now **committed**
        - **Checkpoint:** Write the contents of the update to their final locations within the file system
        - **Free:** some time later, mark the transaction free in the journal by updating the journal superblock
- **Metadata Journaling**
    - Although recovery is now fast through write buffering and write journaling, normal operation of the file system is still slower than we might desire
    - Since, for each write, we are also writing to the journal, the write traffic doubles so it slows down performance
    - Especially painful because sequential write bandwidth is half of its peak bandwidth
        - Between writes to the journal and writes to the main file system, there is a costly seek, which adds a noticeable overhead for some workloads
    - Simpler form of journaling is sometimes called **ordered journaling** (or just **metadata journaling)**, and it is nearly the same, except that **user data is not** written to the journal
    - The data block is instead written to the file system proper, avoiding the extra write, given that most I/O traffic to the disk is data, not writing data twice substantially reduces I/O. Only the metadata (such as inode and bitmap) is written to the log
    - To produce a consistent file system, some data blocks of regular files are written to the disk first, before related metadata is written
    - **1) Data Write:** write data to final location; wait for completion
    - **2) Journal metadata write:** Write the begin block and metadata to the log; wait for the write to complete

- **3) Journal Commit:** Write the transaction commit block (TxE) to the log; wait for the write to complete -> transaction is now committed
- **4) Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system
- **5) Free:** Later, mark the transaction free in journal superblock
- By forcing the data write first, **a file system can guarantee that a pointer will never point to garbage (MAIN PURPOSE)**
- **Core of crash consistency: "write the pointed-to object before the object that points to it"**

- **Tricky Case; Block Reuse**
  - There are some interesting corner cases that make journaling tricky
  - Tweedie Example
    - Suppose you are using some form of metadata journaling (and thus data blocks for files are not journaled)
    - Directory called foo
      - user adds an entry to foo (say by creating a file), and thus the contents of foo are written to the log
    - The user then deletes everything in the directory and the directory itself, freeing up block 1000 use for reuse
    - The user creates a new file called foobar which ends up reusing the same block that used to belong to foo
    - the inode of foobar is committed to disk, as is its data.
    - However, since metadata journaling is in use, only the inode of foobar is committed to the journal; the newly-written data in the block in the file foobar is not journaled

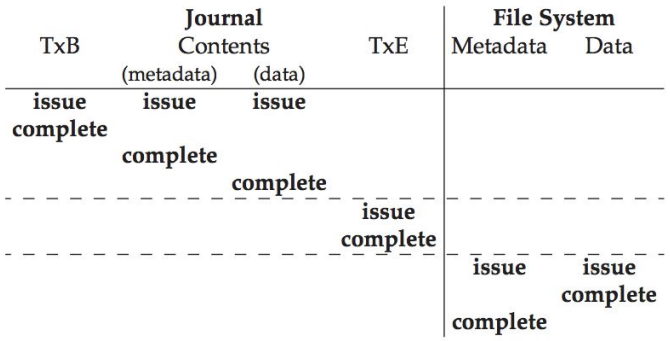


Figure 42.1: **Data Journaling Timeline**

    -
    - However, if a crash occurs and the recovery process simply replays everything, the data in the block could be overwritten with old data
    - Solutions to this problem
      - Never reuse blocks until the delete of said blocks is checkpointed out of the journal
      - What Linux ext3 does instead

        - add a new type of record to the journal called the **revoke record**
    - Deleting the directory would cause a revoke record to be written to the journal.
    - **When replaying the journal, the system first scans for revoke** records, and any such revoked data is never replayed
- **Wrapping Up Journaling: A Timeline**

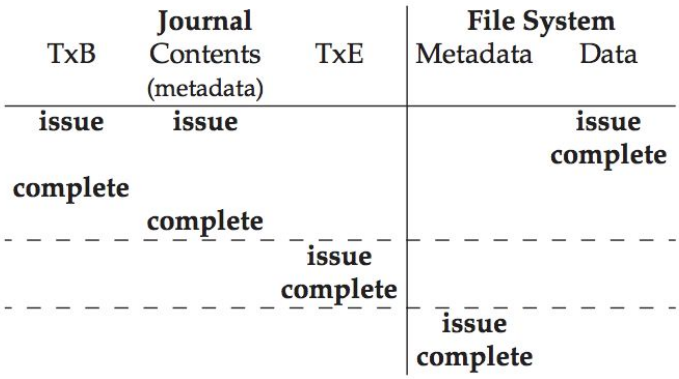


Figure 42.2: **Metadata Journaling Timeline**

-
- **In a real system, completion time is determined by the I/O subsystem, which may reorder writes to improve performance**

42.4: Solution #3: Other Approaches

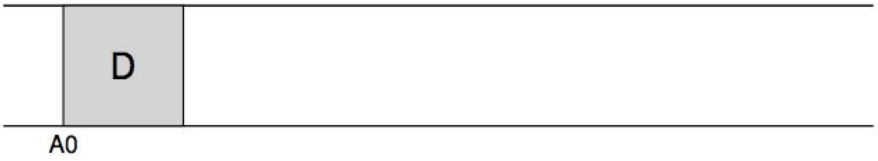
- Soft Updates
    - carefully orders all writes to the file system to ensure that the on-disk structures are never left in an inconsistent state
- Copy-on-write
    - Never overwrites files or directories in place
    - Rather, it places new updates to previously unused location disk
    - After a number of updates are completed, COW file systems flip the root structure of the file system to include pointers to the newly updated structures
- Backpointer-based-consistency BBC
    - no ordering is enforced between writes
    - To achieve consistency, an additional back pointer is added to every block in the system
    - Each data block has a reference to an inode to which it belongs. When accessing a file, the file system can determine if the file is consistent by checking if the forward pointer (e.g. the address in the inode or direct block) points to a block that refers back to it

Chapter 43: Log-Structured File Systems

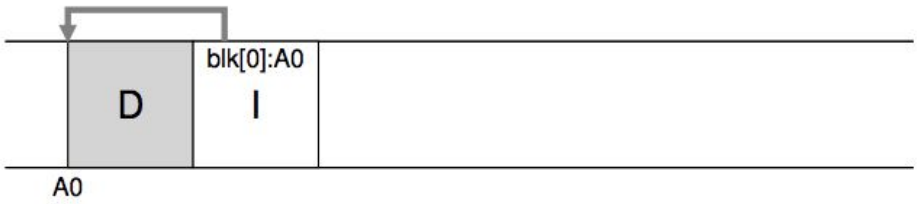
- **System memories are growing:** As memory gets bigger, more data can be caches in memory. As more data is cached, disk traffic increasingly consists of writes -> performance is improved
- **Large gap between random I/O performance and sequential I/O performance:** Hard-drive transfer bandwidth has increased a great deal over the years; as more bits are packed into the surface of a drive, the bandwidth when accessing said bits increases. Using disks in a sequential manner is much more efficient than approaches that causes seeks and rotations
- **Existing file systems perform poorly on many common workloads: FFS would perform a large number of writes to create a new file of size one block**. One for a new inode, one to update the inode bitmap, one to the directory data block that the file is in, one to the directory inode to update it, and one to the new data block that is a part of the new file, and one to the data bitmap to mark that the data block is allocated
- **File systems are not RAID-aware:** Both RAID-4 and RAID-5 have the small-write problem where a logical write to a single block causes 4-physical I/Os to take place
    - An ideal file system would thus focus on write performance, and try to make use of the sequential bandwidth of the disk.
- 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

43.1: Writing to DIsk Sequentially

- How do we transform all updates to file system state into a series of sequential writes to disk?
- Writing the data block to disk might result in the following on-disk layout



- However, when a data block gets written to disk, it is not only the data that gets written to disk; **there is also other metadata** that needs to be updated
    - Write the Inode of the file to disk, and have it point data block D



-
- **This idea of simply writing all updates to the disk sequentially sits at the heart of LFS**

43.2: Writing Sequentially and Effectively

- Writing to disk sequentially is not enough to deliver efficient writes
- You must issue a large number of **contiguous** writes (or one large write) to the drive in order to achieve good performance
- LFS uses **write buffering**: before writing to the disk, LFS keeps track of updates **in memory** and when it receives a sufficient amount of updates, writes them all to disk at once
- The large chunk of updates is known as a **segment**

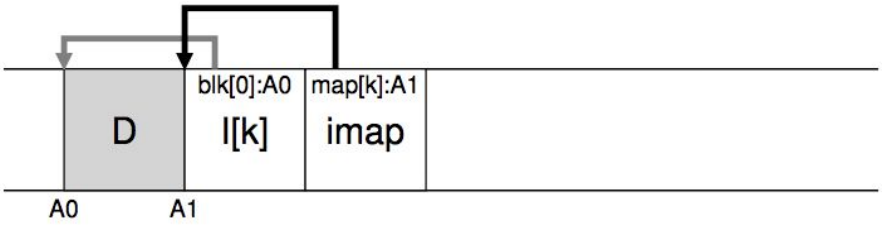
43.3: How much to buffer?

- Depends on the disk -> specifically, **how high the positioning overhead is in comparison to the transfer rate** -> see Chapter 41
- The way to think about it: Every time you write, you pay a fixed overhead of the positioning cost. How much do you have to write in order to amortize that cost?
- The more you write, the better and closer you get to achieving peak bandwidth
- **T(write) = T(position) + D/R(peak)** where D/R(peak) is the time to transfer D, position is the positioning time and write is the time to write
- Effective Rate of Writing: **R(effective) = D/T(write) = D/(T(position) + D/R(peak))**
- **We want to get the effective rate as close as possible to the peak rate**
- **D = F/1-F * R(peak) * T(position)** is the amount we want to transfer/buffer in order to approach peak bandwidth (F is the fraction of the peak rate we want to approach (e.g. 90%/95%)

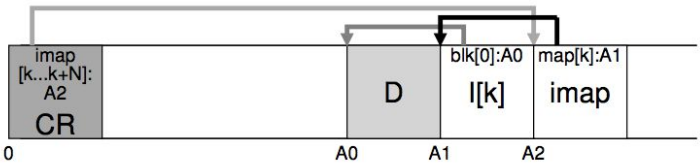
43.4: Problem: Finding Inodes

- To understand how we find an inode in LFS, we must understand how an inode is found in a typical UNIX file system
- In a typical UNIX file system (old) or FFS, finding an inode is easy because they are all arranged in a one-dimensional array based on array-indexing -> use the inode address as an index to the array
- LFS is a difficult because the inodes are scattered throughout the disk, and we can never overwrite in place, and thus the latest version of the inode keeps moving

43.5: Solution Through Indirection: The Inode Map

- **Level of indirection between inode numbers and the inodes through a data structure called the inode map**
    - the imap is a structure t**hat takes an inode number as input and produces the disk address of the most recent version**
    - Downside of indirection -> extra overhead
- The imap must be kept persistent (i.e. written to disk) so that the LFS can keep track of the locations of inodes across crashes
- The imap can be considered an array, with 4 bytes (a disk pointer) per entry
- The LFS places chunks of the inode map right next to where it is writing all of the other new information. Thus, when appending a data block to file k, LFS actually writes the new data block, the inode, and a piece of the inode map all together onto the disk (along with metadata as a pointer)
- 

43.6: Completing the Solution: The Checkpoint Region

- Because the imap is placed in different areas of the disk, we need to find a way to locate the imap
- LFS has a fixed location on disk to begin a file lookup called the **checkpoint region (CR)**
    - Contains pointers (i.e. addresses of) the latest pieces of the inode map, and thus the inode map pieces can be found by reading the CR first. The CR is only updated periodically, and thus performance is not ill-affected
    - 

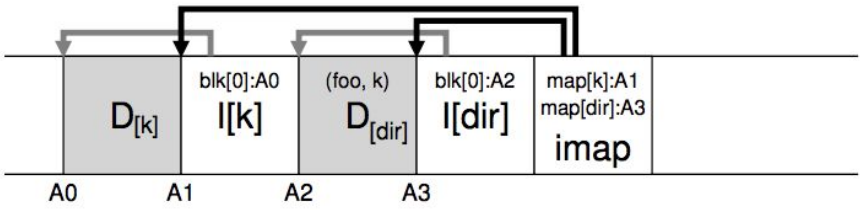
43.7: Reading a File from Disk: A Recap

- Assume we have nothing in memory to begin
- The first on-disk data structure we must read is the checkpoint region. The check point region contains the addresses of (pointers) of the entire inode map, and thus LFS then reads in the inode map and caches it in memory
    - After this point, when given an inode number, LFS simply looks up the inode-number to inode-disk address mapping inthe imap, and reads the most recent version of the inode

- To read a block from the file, LFS proceeds exactly as a typical UNIX file system, by using direct pointers or indirect pointers/doubly indirect pointers
- LFS should perform the same number of I/Os as a typical file system when reading a file from disk; the entire imap is cached and thus the extra work LFS does during a read is to look up the inode's address in the imap

43.8: What about directories?

- Fortunately, directory structure is basically identical to classic UNIX file systems, in that a directory is just a collection of (name, inode number) mappings
- Ex: When creating a file on disk, LFS must both write a new inode, some data, as well as the directory data and the inode that refers to the file



- The piece of the inode map contains the information for the location of both the directory file as well as the newly created file
- First look in the inode map (usually cached in memory) to find the location of the inode of directory dir
- **Recursive Update Problem:** Whenever an inode is updated, its location on disk changes. If we weren't careful, we can have an update to the directory that points to the file, which would then have mandated a change to the parent directory and all the way up the directory tree
- LFS avoids this problem through the inode map because even though the location of an inode may change, the change is never reflected in the directory itself. Rather the imap structure is updated while the directory holds the same name-to inumber mapping

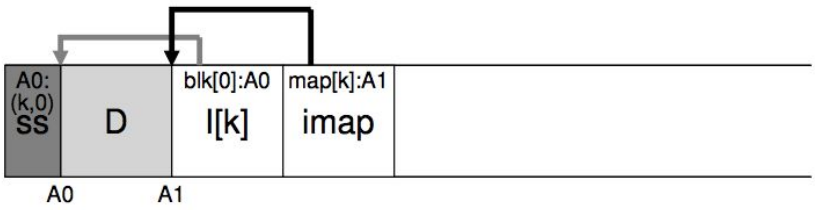
43.9: A New Problem: Garbage Collection

- Old versions of the inodes become garbage
- **Versioning file system:** keeps track of the different versions of a file and allows users to restore old file versions by keeping the older versions
- Segments that hold the data before they are written to disk in large chunks perform a great role in garbage collection
- We can have fragmentation without careful collection as if the garbage collector went through every inode and cleaned up, we won't have large contiguous regions and have **free holes** in allocated space on disk
- Periodically the LFS cleaner reads in a number of old (partially used) segments, determines which blocks are live within the segments, and then write out a new set of segments with just the live blocks within them

43.10: Determining Block Liveness

- Given a data block D within an on-disk segment S, LFS must be able to determine whether D is live
- The LFS includes, for each data block D, its inode number (the file is belongs to) and its file offset (which block of the file this is) -> recorded in the head of a structure as the **segment summary block**
- Look in the summary block and find its inode number (N) and offset (T)
    - Look in the imap to find where N lives and read N from disk
    - Using the offset T, look in the inode to see where the inode thinks the Tth block in the file is on disk
    - If it points to the address it is expecting, the block is still live
    - If not, it is not live and thus can be garbage collected

```
(N, T) = SegmentSummary[A];
inode  = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

- 

-
- Making the process of determining liveness more efficient
    - **Increases its version number when a file is truncated or deleted and records the new version number in the imap**
    - the LFS can short circuit the longer check if the version number is recorded in the on-disk segment

43.11: A Policy Question: Which Blocks to Clean, and When?

- When to clean: either periodically, during idle time, or when you have to because the disk is full
- Determining which blocks to clean: hot segment vs cold segment
    - Hot segment is one in which the contents are being frequently over-written
    - Cold segment: may have a few dead blocks but the rest of its contents are relatively stable

43.12: Crash Recovery and the Log

- What happens if the system crashes while LFS is writing to disk?

    - To ensure that the CR update happens atomically, LFS keeps two CRs, one at either end of the disk, and writes to them alternatively.
    - LFS writes out a header (with timestamp), then the body of the CR, and then finally one last block (also with a timestamp). IF the system crashes during an update, LFS can detect this by seeing an inconsistent pair of timestampes
- On reboot, the LFS can simply read the checkpoint region, the impa pieces it points to, and subsequent files/directories

43.13: Summary

- By never overwriting and simply writing to new unused places in disk, LFS implements an approach called shadow paging that many database systems use

Chapter 44: Flash-based SSDs

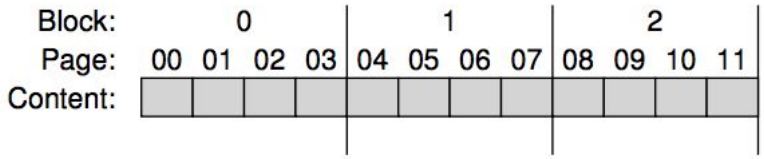
- Generically referred to as **solid state storage,** these devices have no mechanical or moving parts like hard drives; rather, they are simply built out of transistors, much like memory and processors
- Solid-state storage devices retain information despite power loss
- Writing too often to a page will cause it to wear out

44.1: Storing a Single Bit

- **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

44.2: From Bits to Banks/Planes

- 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: which are typically of size 128KB or 256KB
    - Pages: few KB in size (4KB)
- Within each bank there are a large number of blocks; within each block, there are a large number of pages



-

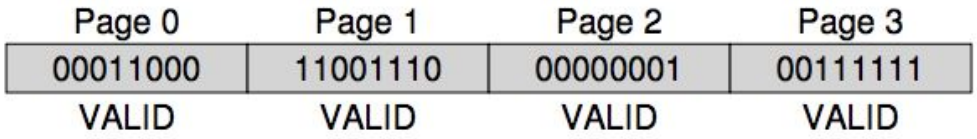
44.3: Basic Flash Operations

- **Read (a page):** A client of the flash chip can read any page (e.g. 2KB or 4KB), simply by specifying the read command and appropriate page number of the device. This operation is typically quite fast, 10s of ms or so -> able to access any location uniformly -> random access device

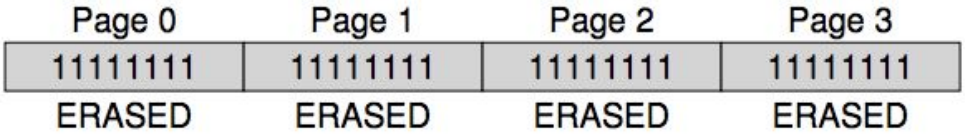
- **Erase (a block): Before writing to a page within the flash**, the nature of the device requires that you first erase the entire block the page lies in
    - Destroys the contents of the block (by setting each bit to 1) -> must be sure that all data that you care about has been copied to somewhere in memory
- **Program (a page):** Once a block has been erased, the program command can be used to change some of the 1's within a page to 0's and write the desired contents of a page to the flash
- Each page has a state associated with it.
    - Pages start in an INVALID state
    - By erasing the block that a page resides within, you s**et the page to ERASED**, which resets the content of each page in the block but also makes them programmable
    - When you program a page, its state is changed to VALID meaning its contents have been set and can be read.
    - Once a page has been programmed, the only way to change its contents is to erase the entire block within which the page resides

44.4: A Detailed Example

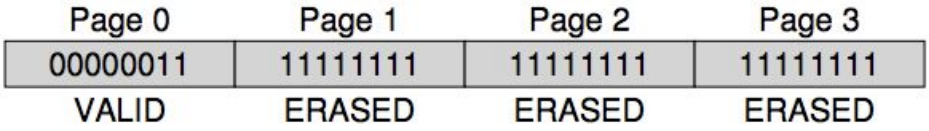
- Because the process of writing is so unusual -> go through example



-
- We wish to write to page 0, filling it with new contents. To write to any page, we must first erase the entire block



-
- We could now go ahead and program page 0, overwriting the old page 0 as desired



-
- Thus, before overwriting any page within a block, we must first move any data we care about to another location (e.g. memory or somewhere on the flash)

44.4: Flash Performance and Reliability

- Read latencies of raw flash chips are very good, taking 10s of microseconds to complete
- Program latency is higher and more variable
- Erases are quite expensive and take a few milliseconds typically

- Flash chips are pure silicon and have very little reliability issues that we have to worry about
- Reliability Issues
    - Wear out: Each block has can be erase and programmed 10,000 times before falling
    - **Disturbance:** 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**

44.5: From Raw Flash to Flash-Based SSDs

- The standard storage interface is a simple block-based one, where blocks of size 512 bytes can be read or written, given a block address
- Task of flash-based SSD is to provide that standard block interface atop the raw flash chips inside of it
- SSD contains some number of raw flash chips. Also contains some amount of volatile (non-persistent) memory (e.g. SRAM)
    - such memory is useful for caching and buffering of data as well as for mapping tables
    - Contains control logic to orchestrate device operation
- **Flash translation layer (FTL):** accomplishes the task of turning logical blocks into underlying physical blocks/pages.
- Key: Use multiple flash chips in parallel
- **Write amplification:** defined as the total write traffic (in bytes) issued to the flash chips by the FTL divided by the total write traffic issued by the client to the SSD
- How to achieve high reliability
    - The FTL should try to spread out writes across the blocks of the flash as evenly as possible, **ensuring that all of the blocks of the device wear out at roughly the same time**
    - **Wear leveling**
- **Program disturbance: FTLs commonly program pages within an erased block in order from low to high page**

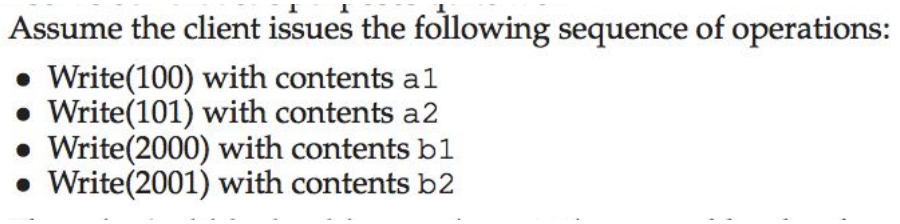
44.6: FTL Organization: A Bad Approach

- Read to logical page N is mapped directly to a read of physical page N (direct mapped)
- A write to logical page N is more complicated
    - FTL first has to read the entire block that page N is contained within -> erase the block -> then program
- Reliability and performance are both bad -> wears out fast + inefficient

44.7: 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 -> **logging**

- To allow for subsequent reads of block N, the device keeps a mapping table that stores the physical address of each logical block in the system


Assume the client issues the following sequence of operations:

- Write(100) with contents a1
- Write(101) with contents a2
- Write(2000) with contents b1
- Write(2001) with contents b2

- These logical block addresses are used by the client of the SSD (e.g. file system) to remember where information is being located
- Internally, the device must transform these block writes into the erase and program operations supported by the raw hardware
- For each logical block address, must record physical page of the SSD stores its data
- What if the client wants to read logical block 100? How can it find where it is?
    - The SSD must transform a read issues to logical block 100 into a read of physical page 0
    - **Records the fact that when logical mapping writes to physical -> logged in in-memory mapping table**
    - SSD finds a location for the write, usually just picking the next free page to avoid disturbance; then programs that page with the block's contents, and records the logical-to-physical page number required to read the data
- This logical-to-physical page mapping is used later to improve performance ->kinda like a cache
- Log-based approach by its nature improves performance (erases only being required once in a while, and the costly read-modify-write of the direct-mapped approach avoided altogether
- Overwrites of logical blocks lead to garbage -> older versions of data around the drive and taking up space
- Has to perform **garbage collection** to find said blocks and free space for future writes.
- High cost of in-memory mapping tables -> the larger the device, the more memory such tables need

44.8: Garbage Collection

- Garbage is created so that garbage collection is necessary
- Although a page may be considered VALID it may still have garbage in them
- Find a block that contains one or more garbage pages, read in the live pages from the block, write out those live pages to the log, and reclaim the entire block for use in writing
- For the garbage collector to function, there must be enough information within each block to enable the SSD to determine whether each page is live or dead
    - One way to achieve this end is to store, at some location within each block, information about which logical blocks are stored within each page
    - The device can then use the mapping table to determine whether each page within the block holds live data or not

- Garbage collection can be expensive, requiring reading and rewriting of live data. The ideal candidate for reclamation is a block that consists of only dead pages
- **To reduce GC costs**, some SSDs overprovision the device by adding extra flash capacity, cleaning can be delayed and pushed to the **background,** perhaps done at a time when the device is less busy

44.9: Mapping Table Size

- The second cost of log-structuring is the potential for extremely large mapping tables. Thus, this page-level FTL scheme is impractical
- **Block-Based Mapping**
    - One way to reduce the costs of mapping is to only keep a pointer per block of the device, instead of per page, reducing the amount of mapping information by a factor of Size(block)/Size(page)
    - A block-based mapping inside a log-based FTL does not work very well for performance reasons
    - “Small write problem”: FTL must read a large amount of live data from the old block and copy it into the new one
        - Data copying increases overhead and decreases performance drastically
    - Use chunk numbers and offsets to refer to the blocks (chunks) and page numbers (offset)
    - Reading in a block-based FTL is very easy
        - **FTL extracts the chunk number from the logical block address presented by the client, by taking the topmost bits out of the address**
        - FTL computes the address of the desired flash page by adding the offset from the logical address to the physical address
- **Hybrid Mapping**
    - The FTL keeps a few blocks erased and directs all writes to them
        - these blocks **are called the log blocks**
    - The FTL wants to be able to write any page to any location within the log block without all the copying required by a pure block-based mapping
        - it keeps per-page mappings for these log blocks
    - FTL thus logically has two types of mapping table in its memory -> a small set of per-page mappings in what we’ll call the log table, and a larger set of per-block mappings in the data table
    - **The key to the hybrid mapping strategy is to keep the number of log blocks small**
        - has to periodically examine log blocks and switch them into blocks that can be pointed to by only **a single block pointer**
    - **Switch merge: Switches the contents of the blocks onto the log block if they are written in the same order**
    - **Partial Merge and Full Merge require more I/O and thus are not ideal for the FTL**

44.10: Wear Leveling

- Because multiple erase/program cycles will wear out a flash block, the FTL should try its best to spread that work across all the blocks of the device evenly
- If a block has long-lived data that is never overwritten, then the garbage collection will never be called
- Solution to above problem
    - FTL must periodically read all the live data out of such blocks and re-write it elsewhere -> make the block available for writing again
- This decreases performance as extra I/O is required to ensure that all blocks wear at roughly the same rate

44.11: SSD Performance and Cost

- **Performance**
    - Unlike hard disk drives, flash-based SSDs have no mechanical components, and in fact are in many ways similar to DRAM (random access)
    - Performance of SSDs greatly outstrips modern hard drives, even when performing sequential I/O
- **Cost**
    - **Very expensive (60c per GB)** so many systems use a combination of hard drives and SSDs

Chapter 45: Data Integrity and Protection

- How should systems ensure that the data written to storage is protected?

45.1: Disk Failure Modes

- Modern disks also have latent-sector errors (LSEs) and block corruption where they have difficulties accessing certain sectors and blocks
- **LSEs** arise when a disk sector/group of sectors have been damaged in some way. If the head touches the surface for some reason (head crash), it may damage the surface and make the bits unreadable
- **Error-correcting codes (ECC)** are used by the drive to determine whether the on-disk bits in a block are good, and in some cases, to fix them.
- **Block corruption:** Buggy disk firmware may write a block the wrong location; the disk ECC indicates the block contents are fine, but from the client's perspective, the wrong block is returned when subsequently accessed
    - Block may get corrupted when it is transferred from the host of the disk across a faulty bus
    - these kinds of faults are called **silent faults** because the disk gives no indication of the problem when return the faulty data
- Partial-faults: System seems to work but in actuality, some bits may be inaccessible/corrupted
- Findings about LSEs

    - Costly drives with more than one LSE are as likely to develop additional errors as cheaper drives
    - For most drives, the annual error rate increases in year 2
    - Number of LSEs increase with disk size
    - Most disks with LSes have 50
    - Exists a significant amount of spatial/temporal locality
- Findings about corruption
    - Chance of corruption varies greatly across different drive models
    - Age effects are different across models
    - Workload and disk size have little impact on corruption
    - Weak correlation with LSEs

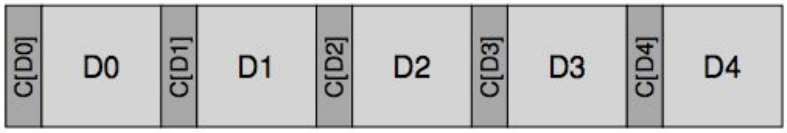
45.2: Handling Latent Sector Errors

- When a storage system tries to access a block, and the disk returns an error, the storage system should simply use whatever redundancy mechanism it has to return the correct data
- In a mirrored RAID, the system should access another copy. In RAID-4 or 5 the system should use the blocks to reconstruct the parity block
- What happens if we encounter an LSE during reconstruction?
    - Some systems add an extra degree of redundancy
    - RAID-DP: equivalent of two parity disks instead of one

45.3: Detecting Corruption: The Checksum

- How can we prevent users from getting bad data when corruption arises?
- Unlike latent sector errors, the **detection of corruption is the main issue**
- Once it is known that the particular block is bad, the fixing process is the same as for latent sector errors
- **Checksum:** Simply the result of a function that takes a chunk of data as input and and computes a function over said data -> produces small summary of contents of data
- **Common Checksum Functions**
    - Tradeoff: the more protection you get, the costlier it is
    - **XOR based checksums**: XOR each chunk of the data block being checksummed and produce a single value that represents the XOR of the entire block
        - Reasonable but has its limitations
        - If two bits in the same position within each checksummed unit change, the checksum will not detect corruption
    - **Addition:** Fast because computing it just requires performing 2's complement addition over each chunk of the data -> ignores overflow
        - Not good if the data is shifted
    - **Fletcher checksum:** Assume a block D consists of bytes d1 … dn
        - s1 = (s1+d)%255, s2=(s2+s1)%255, and so on

    - **Cyclic Redundancy Check (CRC):** Compute the checksum over a data block D. Treat D as if it is a large binary number and divide it by an agreed upon value k
        - Remainder of the division is the value of the CRC
    - No perfect checksum because it is possible that two data blocks with non-identical contents will have identical checksums -> **collision**
- **Checksum Layout**



    - Because checksums are usually small (around 8 byes), and disks only can write in sector-sized chunks (512 bytes) one problem we have is how we are going to achieve the above layout

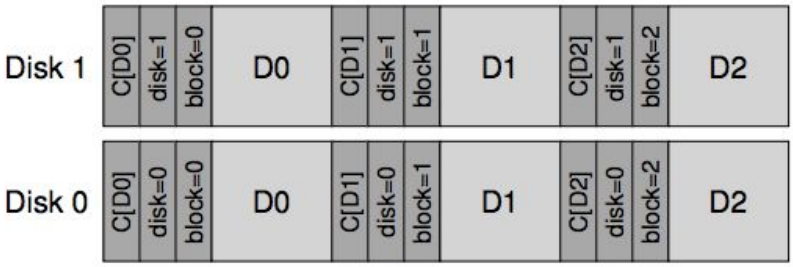
    - The n checksums are stored together in a sector, followed by n data blocks
        - followed by n checksums for the next n blocks and so on
    - This layout can be less efficient than the first layout

45.4: Using Checksums

- When reading a block D, the client (i.e. file system or storage controller) also reads its checksum from disk C(D), which we called the **stored checksum**
- Client compares the stored and computed checksum over the retrieved block D, and sees if they are equal. If the data does not match, we have a corruption!
- You can either return an error if we don't have redundant copies, or we can use the redundant copy

45.5: A New Problem: Misdirected Writes

- Misdirected writes arise in disk and RAID controllers which write the data to disk correctly, except they are written to the wrong location
- Solution: Add a little more information to the checksum
    - **physical ID:** if the stored information now contains the checksum C(D) as well as the disk and sector number of the block, it is easy for the client to determine whether the correct information resides within the block



-

45.6: One Last Problem: Lost Writes

- Lost writes occur when the device informs the upper layer that a write has completed but in fact it never persisted
- **Write verify/read-after-write:** read back the data after a write -> system can ensure that the data indeed reached the disk surface

Lecture Notes

Maximizing Cylinder Locality

- Seek time dominates the cost of disk I/O
    - greater than or equal to rotational latency
    - and much harder to optimize by scheduling
- Live systems do random access disk I/O
    - directories, i-nodes, programs, data, swap space
    - all of which are spread all across the disk
- but the access is not uniformly random
    - 5% of the file account for 50% of the disk access
    - users often operate in a single directory
- Creates lots of mini-file systems
    - each with grouped inodes, directories, data
    - significantly reduce the mean-seek distance

Disk Seek/Latency Scheduling

- Deeper queues mean more efficient I/O
    - elevator scheduling of seeks
    - choose multiple blocks in the same cylinder
- Consecutive block allocation helps
    - more requests can be satisfied in a single rotation
- works whether scheduling is in OS or drive
    - but the drive knows the physical geometry
    - drive can accurately compute see/rot times

Allocation/Transfer Size

- Per operation overheads are high
    - DMA startup, seek, rotation, interrupt service
- larger transfer units more efficient
    - amortize fixed per-op costs over more bytes/op
    - multi-megabytes transfers are very good
- this requires space allocation units
    - allocate space to files in much larger chunks
    - large fixed-size chunks -> internal fragmentation
    - therefore we need variable partition allocation

Block Size vs Internal Fragmentation

- Large blocks are a performance win
    - fewer next-block lookup operations
    - fewer, larger I/O operations
- Internal fragmentation rises w/block sizes
    - mean loss = block size / (2* mean file size)
- Can we get the best of both worlds?
    - most blocks are very large
    - the last block is relatively small
- All the blocks of a file are a certain block size -> but set the last block size
    - Idea: Use big blocks for everything except for the last one

I/O Efficient DIsk Allocation

- Allocate space in large, contiguous extents
    - few seeks, large DMA transfers
- Variable Partition disk allocation is difficult
    - many files are allocated for a very long time
    - space utilization tends to be high (80-90%)
    - special fixed size free-lists don’t work as well
- external fragmentation eventually wins
    - new files get smaller chunks -> farther apart
    - file system performance degrades with age

Read Caching

- Disk I/O takes a very long time
    - deep queues, large transfers improve efficiency
    - they do not make it significantly faster
- we must eliminate much of our disk I/O
    - maintain an in-memory cache
    - depend on locality, reuse of the same blocks
    - read-ahead (more data than requested) into cache
    - check cache before scheduling I/O

- all writes must go through the cache
    - ensure it is up-to date

Read-Ahead

- Request blocks before they are requested
    - store them in cache until later
    - reduces wait time, may improve disk I/O
- When does it make sense?
    - When client specifically requests sequential access
    - when client seems to be reading sequentially
- What are the risks?
    - may waste disk access time reading unwanted blocks
    - may waste buffer space on unneeded blocks

Special Purpose Caches

- Often block caching makes sense
    - files that are regularly processed
    - indirect blocks that are regularly referenced
- Consider i-nodes (32 per 4K block)
    - only recently used I-nodes likely to be reused
- consider directory entries (256 per 4K blocks)
    - 1% of entries account for 99% of access
- perhaps we should cache entire paths

Special Caches - Doing the Math

- consider the hits per byte per second ratio
    - 2 hits/4K block
    - 1 hits/32 byte dcache entry
- Consider the savings from extra hits
    - e.g. block reads/second * 1.5ms/read = 67ms
- consider the cost of the extra cache lookups
    - e.g. 1000 lookup/s * 50ns per lookup = 50us
- Consider the cost of keeping cache up to date
    - e.g. 100 upd/s*150ns per upd = 15us
- net benefit: 75ms -65 us = 74.935ms/s

When can we outsmart LRU?

- Is it hard to guess what programs will need
- Sometimes we know what we won't re-read
    - load module/DLL read into a shared segment
    - an audio/video frame that was just played
    - a file that was just deleted or overwritten
    - a diagnostic log file

- dropping these files from the cache is a win
    - allows a longer file to the data that remains there

Write-Back Cache

- Writes go into a write-back cache
    - They will be flushed out to disk later
- With a write through cache -> the data will go through fairly quickly
- We can expect more data to be written -> buffer so that when we write it out the write is full -> **Lazy write**
- Aggregate small writes into large writes
    - if application does less than full block writes
- Eliminate moot writes
    - if application subsequently rewrites same data
    - if application subsequently deletes the file
- accumulate large batches of writes
    - a deeper queue to enable better disk scheduling

Persistence vs Consistency

- POSIX Read-After-Write Consistency
    - any read will see all prior-writes
    - even if it is not the same open file instance
- Flush-on-Close Persistence
    - write(2) is not persistent until close(2) or fsync(2)
    - think of these as commit operations
    - close(2) might take a moderately long time
- This is a compromise
    - strong consistency for multi-process applications
    - enhanced performance from write-back cache

File Systems-Device Failures

- Unrecoverable Read Errors
    - signal degrades beyond ECC ability to correct
    - background scrubbing can greatly reduce
- mis-directed or incomplete writes
    - detectable w/independent checksums
- complete mechanical/electronic failures
- all are correctable w/redundant copies
    - mirroring, parity, or ensure coding
    - individual block of whole volume recovery

***LOOK AT BATCHED JOURNAL ENTRIES