Advanced File Systems

Operating Systems Design

Index

- Berkeley FFS
- Log-structured FS
- Fsck and journaled FS
- Google FS

Bibliography

- Crash Consistency: FSCK and Journaling
  - “Fsck - The UNIX File System Check Program” Marshall Kirk McKusick and T. J. Kowalski
  - http://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf
- Berkeley FFS
  - Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry (August 1984). "A Fast File System for UNIX" (PDF). ACM Transactions on Computer Systems. 2 (3): 181–197.
  - http://pages.cs.wisc.edu/~remzi/OSTEP/file-ffs.pdf
- LFS
  - Rosenblum, Mendel; Ousterhout, John K (February 1992), "The Design and Implementation of a Log-Structured Filesystem", ACM Transactions on Computer Systems, 10 (1): 26–52
  - http://pages.cs.wisc.edu/~remzi/OSTEP/file-lfs.pdf
- GFS
  - Ghemawat, S.; Gobioff, H.; Leung, S. T. (2003). "The Google file system". Proceedings of the nineteenth ACM Symposium on Operating Systems Principles - SOSP '03

Index

- **Berkeley FFS**
  - **Design goal: increase reliability and performance**
- Log-structured FS
- Fsck and journaled FS
- Google FS

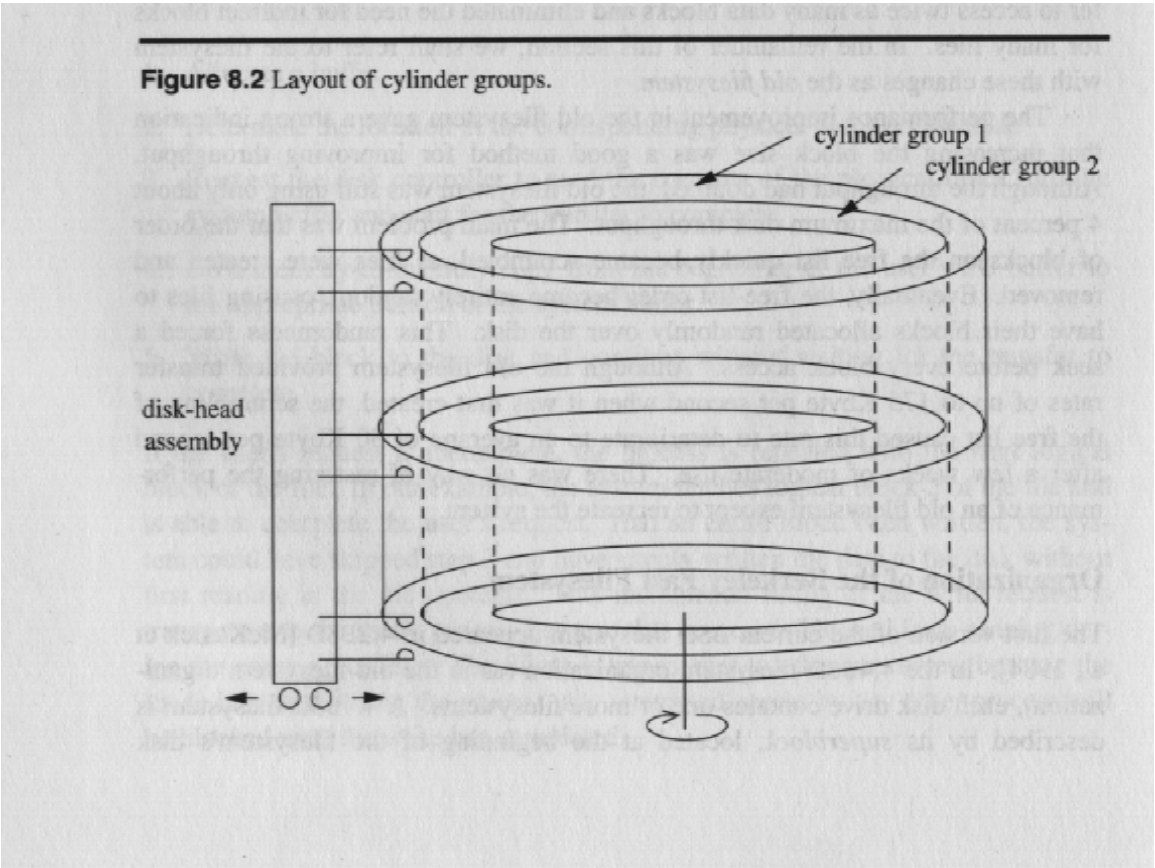
Classical Unix File System

- Traditional UNIX file system keeps I-node information separately from the data blocks
  - Accessing a file involves a long seek from i-node to the data.
- Since files are grouped by directories
  - When I-nodes for the files from the same directory are not allocated consecutively, non-consecutive block reads are long.
  - Allocation of data blocks is sub-optimal because often next sequential block is on different cylinder.
- Reliability: only one copy of super-block, and i-node list;
- Free blocks list quickly becomes random, as files are created and destroyed: seek operations
- Due to small block size (512-1024), many random blocks, many reads, files too small, low bandwidth utilization on system bus.

Berkeley Fast File System (FFS)

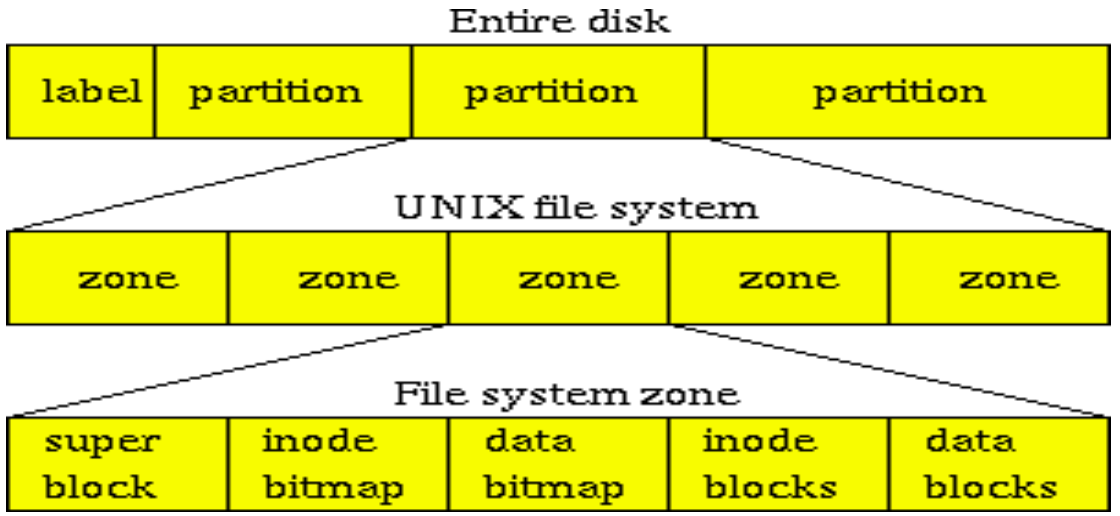
- BSD Unix did a redesign in mid 80s that they called the Fast File System (FFS)
- Improved disk utilization, decreased response time
- Minimal block size is 4096 bytes allows to files as large as 4G to be created with only 2 levels of indirection.
- Disk partition is divided into one or more areas called *cylinder groups;*

Cylinder Groups



Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry (August 1984). "A Fast File System for UNIX" (PDF). ACM Transactions on Computer Systems. 2 (3): 181–197.

File system structures



Data Block and inode placement

- The original Unix FS had two placement problems:

  Data blocks allocated randomly in aging file systems

  - Blocks for the same file allocated sequentially when FS is new

  - As FS “ages” and fills, need to allocate into blocks freed up when other files are deleted

  - Deleted files essentially randomly placed So, blocks for new files become scattered across the disk.

  INODEs allocated far from data blocks

  - All inodes at beginning of disk, far from data

  - Traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks

- FRAGMENTATION. Both of these problems generate many long seeks (100X slower than sequential read)

Cylinder Groups

- BSD FFS addressed these problems using the notion of a ***cylinder group***

  - Disk partitioned into groups of cylinders

  - Data blocks in same file allocated in same cylinder group

  - Files in same directory allocated in same cylinder group

  - Inodes for files allocated in same cylinder as file data blocks

  - Minimize seek times

- Free space requirements

  - To be able to allocate according to cylinder groups, the disk must have free space scattered across cylinders

  - 10% of the disk is reserved just for this purpose (root)

  - Only used by root – why it is possible for “df” to report >100%. And why a filesystem is “full” at 90%.

Cylinder Groups

- Each cylinder group is one or more consecutive cylinders on a disk.
- Each cylinder group contains a redundant copy of the super block, and I-node information, and free block list pertaining to this group.
- Each group is allocated a static amount of i-nodes.
- A cylinder group keeps i-nodes of files close to their data blocks to avoid long seeks; also keep files from the same directory together.

File System Block Size

- Problem: Small blocks (1K) cause two problems:

  -Low bandwidth utilization

  - Small max file size (function of block size)

  FFS Solution: use a larger block (4K).

  - Very large files, only need two levels of indirection for 2^32

  - Improved bandwidth. As block size increases the efficiency of a single transfer also increases, and the space taken up by i-nodes and block lists decreases.

File System Block Size

- Problem: as block size increases, the space is wasted due to internal fragmentation.

  FFS Solution: introduce “fragments” (1K pieces of a block)

  - Divide block into *fragments*;
  - Each fragment is individually addressable;
  - Fragment size is specified upon a file system creation;
  - The lower bound of the fragment size is the disk sector size;

- Problem: Media failures

  FFS Solution: Replicate superblock, subdivide INODE list, directory entries among cylinder groups

FFS Considerations

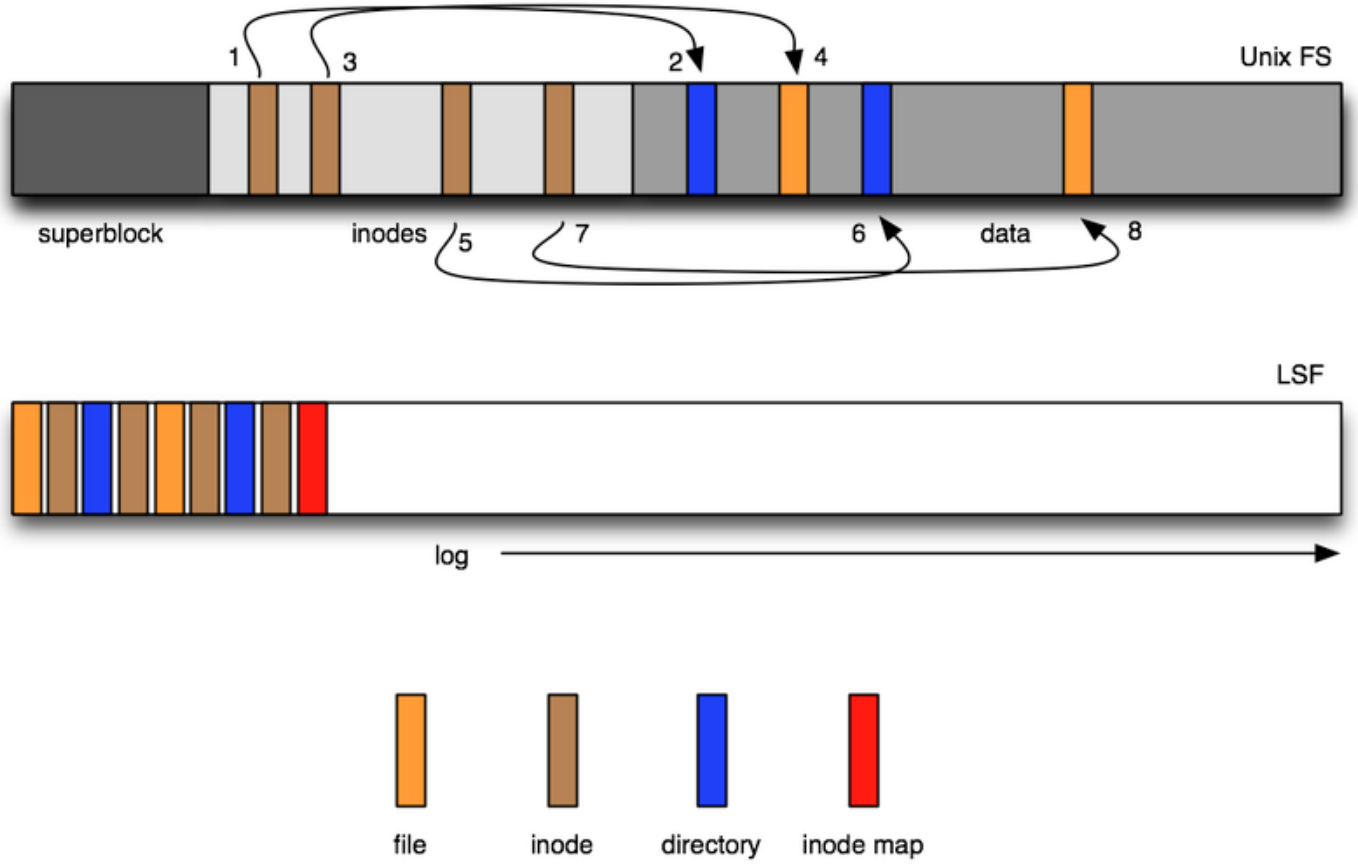
- **Problems with FFS:**
  - With cylinder groups or extent allocation, data placement is improved, but still have many small seeks possibly related files are physically separated
  - inodes separated from files (small seeks)
  - directory entries separated from inodes
  - Metadata requires synchronous writes: with small files, most writes are to metadata (synchronous) and synchronous writes very slow;

- **Solution: Log-Structured File System**

Index

- Berkeley FFS
- **Log-structured FS**
  - **Design goal: increase the performance of small write operations**
- Fsck and journaled FS
- Google FS

Comparison Unix FS with Log-Structured FS



http://work.tinou.com/2012/03/log-structured-file-system-for-dummies.html

Logging File System (LFS)

- Logging File Systems Developed by Ousterhout & Douglis (1992)
- LFS treats the disk as a single log for appending
- All info written to disk is appended to log
  - Data blocks, attributes, inodes, directories, etc.
- LFS collects writes in disk cache, write out entire collection in one large disk request
  - Leverages disk bandwidth
  - No seeks (assuming head is at end of log)

FFS Problems – LFS Solutions

- Problem: small writes cause bad disk access performance.

  Solution: Logging, each write for both data and metadata is appended to the end of the sequential log.

FFS Problems – LFS Solutions

- Problem: FFS uses inodes to locate data blocks

  - Inodes pre-allocated in each cylinder group

  - Directories contain locations of inodes

  LFS appends inodes to end of the log just like data. But, it makes them hard to find.

  Solution:

  - Use another level of indirection: Inode maps

  - Inode maps map files to inode location (on various segments on the disk)

  - Location of inode map blocks kept in a checkpoint region

  - Checkpoint region has a fixed location

  - Cache inode maps in memory for performance

LFS Problems, Solutions

- LFS has two challenges it must address for it to be practical:

  1) Locating data written to the log.

  - FFS places files in a location, LFS writes data “at the end of the log”

  2) Managing free space on the disk

  - Disk is finite, so log is finite, cannot always append. LFS append-only quickly runs out of disk space

  - Need to recover deleted blocks in old parts of log

LFS Problems, Solutions

- Approach:
  - Fragment log into segments
  - Segments can be anywhere on disk
  - Reclaim space by cleaning segments
    - Read segment
    - Copy live data to end of log
    - Now have free segment you can reuse

- Memory / disk cleaning (garbage collection) is a big problem
  - Costly overhead

Index

- Berkeley FFS
- Log-structured FS
- **Fsck and journaled FS**
  - **Design goal: consistency and performance of reconstruction**
- Google FS

FS crash consistency

- One major challenge faced by a file system is how to update persistent data structures despite the presence of a power loss or system crash
- Example
  - two on-disk structures, A and B in order to complete a particular operation.
  - either A or B reaches disk first
  - system crashes in-between
  - INCONSISTENT state: how to solve?
- Approaches:
  - traditional: fsck
  - new: journaling (write-ahead logging)

Crash consistency - example

- append a block to a file
- open + lseek + write
- 8 inodes + 8 data blocks
- Inode 2 I[v1] is allocated and data block 4
- I[v1] means inode version 1, B[v1] data block bitmap version 1

	v1	v2 (after append)
I	owner : remzi permissions : read-write size : 1 pointer : 4 (Da) pointer : null pointer : null Pointer : nulll	owner : remzi permissions : read-write size : 2 pointer : 4 (Da) pointer : 5 (Db) pointer : null pointer : null
B	0001000	0001100

- To get v2 on the disk we need three disk operations: I[v2], B[v2] and Db
- But the FS may cache them in the buffer cache or page cache and write them later
- If a crash happens before strange things can happen

Arpaci-Duseau

Crash scenarios

- Just one disk write performed, the other two are lost
  - Db: as if the write never was done (consistent system)
  - I[v2]:
    - pointer points to garbage data
    - inconsistency: data bitmap says Db has NOT been allocated but the inode says it has been allocated
  - B[v2]
    - inconsistency: data bitmap says Db has been allocated but the inode says it has NOT been allocated
    - space leak

Crash scenarios (cont.)

- Two disk writes performed, one is lost
  - I[v2] and B[v2]:
    - consistent file system
    - garbage is read
  - I[v2] and Db
    - pointer points to write data
    - inconsistency: data bitmap says Db has NOT been allocated but the inode says it has been allocated
  - B[v2] and Db:
    - inconsistency: data bitmap says Db has been allocated but the inode says it has NOT been allocated

fsck

- We would like to atomically perform the three writes, but not possible: CRASH CONSISTENCY PROBLEM
- Solution 1: FS checker
- fsck: runs before the FS is started
- Superblock
  - sanity checks making sure the superblock has not been corrupted
  - e.g.: FS size > number of allocated blocks
- Free blocks
  - scans inodes, indirect blocks, double indirect blocks to understand which blocks are allocated
  - if inconsistency between blocks and inodes, it trusts the inodes
  - all inodes that look in-use are marked correctly in the inode bitmap

fsck (cont.)

- Inode state
  - the type field is correct (e.g., file, directory).
  - if suspect, the inode is cleared
- Inode links
  - verify link counts by making an own counting
  - if zero references, move them to lost+found
- Duplicates
  - duplicate pointers: two inodes pointing to the same data block
  - either clear one pointer
  - or copy the block
  - bad blocks: pointers outside partition size
- Directory checks
  - . and .. are the first entries
  - all inodes are allocated
  - directory linked once in the hierarchy

Write-ahead logging (journaling)

- How to avoid the long times necessary by fsck?
- before doing the three writes, write on the disk a note about what you are about to do (write-ahead)
- write it to a log
- if a crash occurs in between writes, you know what it has failed
- you go to the log and try again
- no need to scan the whole file system again
- JFS examples
  - Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, and Windows NTFS.

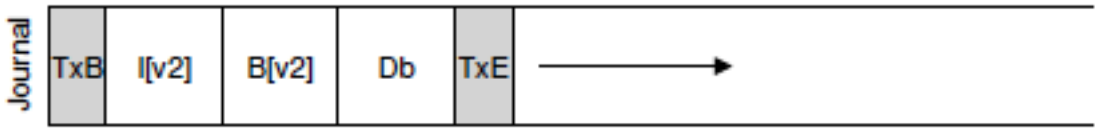
ext3

- Disk divided into several block groups
- Journal after superblock or on another device

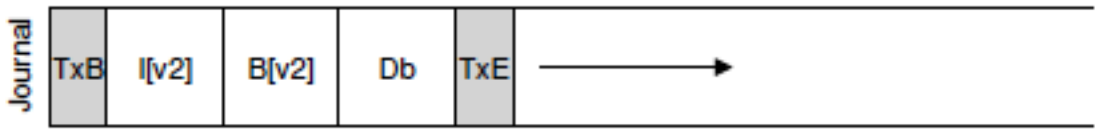
- Example from before
  - write the inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk
  - Before writing them to their final disk locations, we are now first going to write them to the log (journal)

  - TxB - transaction begin: addresses of blocks l[v2], B[v2], Db + transaction ID
  - TxE- transaction end: marker of the end of transaction

1. Journal write (above)
2. Checkpoint: write the data and metadata to disk

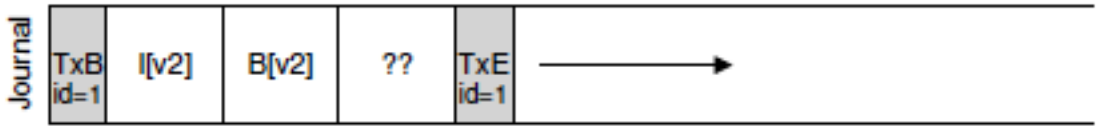


Crash during journal write



1. Journal write: write the 5 blocks to the disk
2. Checkpoint: write the data + metadata to disk

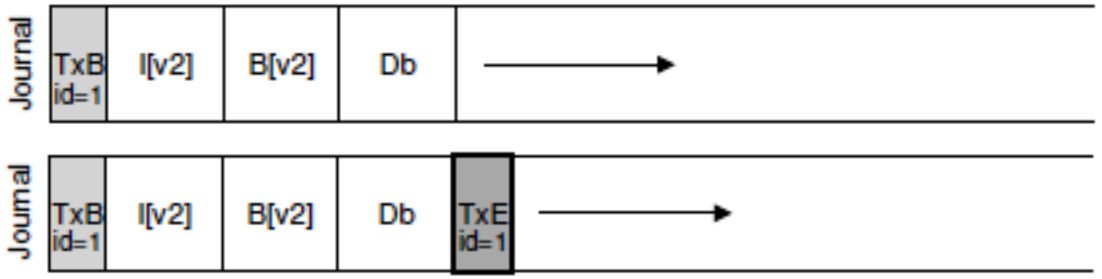
- How to address crash during journal write
  - Write the 5 blocks one by one, waiting for each to complete: slow
  - Ideally write all 5 blocks at once sequentially
    - But disk may schedule them in a different order, e.g. TxB, I[v2], B[v2], TxE, Db
    - If crash occurs before writing Db: correct log entry but pointing to garbage



Arpaci-Duseau

Crash during journal write (2)

- Solution: two steps transactions



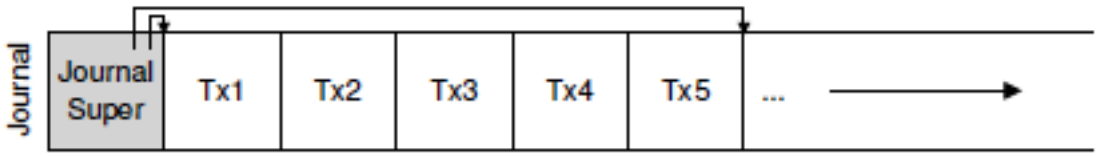
1. Journal write: write the first 4 blocks to disk
2. Journal commit: write The TxE block to disk
3. Checkpoint: write the data + metadata to disk

Recovery

- Crash before step 2: skip the transaction
- Crash after step 2:
  - When rebooting look in the log for transactions that have committed to disk
  - Replay them in order
  - The disk structures are now consistent: mount the FS
  - Even though some updates of metadata/data are replayed again, everything is fine!

Optimizations

- One transaction at a time too much disk traffic
  - Batch transactions into a mega-transaction and commit it to the disk after some seconds
- Making the log finite
  - If log too long, the recovery time too long
  - If log full, no new transactions can be committed
  - Circular log: free the space of each checkpointed transaction
    - E.g.. Use a Journal superblock for marking the window of non-checkpointed transactions



1. Journal write: write the first 4 blocks to disk
2. Journal commit: write the TxE block to disk
3. Checkpoint: write the data + metadata to disk
4. Free: later mark the transaction free by updating journal superblock

Classifications

- Classifications
  - Physical logging: Log the content of the metadata and data (previous examples)
  - Logical logging
    - More compact logical representation
    - E.g., “this update wishes to append data block Db to file X”
    - Space saving, Better performance
- Data vs. metadata
  - Data journaling: log the data into the journal (previous examples)
  - Metadata (ordered) journaling: same as data, but data not logged but just written to the final location first

1. Write data to final location and wait for completion (this guarantees a pointer does not point to garbage)
2. Journal write: write the first 4 blocks to disk
3. Journal commit: write the TxE block to disk
4. Checkpoint metadata to final disk location
5. Free: later mark the transaction free by updating journal superblock

Index

- Berkeley FFS
- Log-structured FS
- Fsck and journaled FS
- **Google FS**
  - **Design goals: scalability, performance, use of in-expensive hardware, reliabilty and availability**

Introduction and Motivation

- Background:
  - Goals of distributed file system:
    - Performance, scalability, reliability, and availability.
- Motivation:
  - Departure from some earlier file system design assumptions based on *Google's application workloads and technological environment*.

Different point(1/3)

- Component failures are the norm rather than the exception.
  - The file system consists of hundreds or even thousands of storage machines built from **inexpensive commodity parts.**
  - Things to do:
    - Constant monitoring
    - Error detection
    - Fault tolerance
    - Automatic recovery

Different point(2/3)

- Files are huge by traditional standards(Multi-GB files are common):
  - Each file typically contains many application objects such as web documents.
    - **Working with fast growing data sets of many TBs comprising billions of objects.**
    - **It is unwieldy to manage billions of traditionally KB-sized files.**
  - Things to do:
    - Redesign I/O operation and block sizes.

Different point(3/3)

- **Most files are mutated by appending new data** rather than overwriting existing data.
  - Random writes within a file are practically non-existent.
  - Once written, the files are only read, and often only sequentially.
    - E.g. data streams continuously generated by running applications
    - E.g. intermediate results produced on one machine and processed on another

Six Design Assumption

1. The system is built from many inexpensive commodity components that often fail.
2. The system stores a number of large files.
   - Small files must be supported, but we need not optimize for them.
3. The workloads primarily consist of two kinds of reads:
   - Large streaming reads and small random reads.
   - Performance-conscious applications often batch and sort their small reads.

Six Design Assumption(con't)

4. The workloads also have many large, sequential writes that append data to files.
   - Once written, files are seldom modified again.
5. The system must **efficiently** implement well-defined semantics for **multiple clients that concurrently append to the same file**.
6. High sustained bandwidth is more important than low latency.
   - Most of our target applications place a premium on processing data in bulk at a high rate.

GFS Architecture

- Roles:
  - A GFS cluster consists of a single *master* and multiple *chunk servers* and is accessed by multiple *clients.*
- Files are divided into fixed-sized *chunks.*
  - Each chunk is identified by a globally unique 64 bit chunk handle.
    - Assigned by the master at the time of chunk creation.

Overview

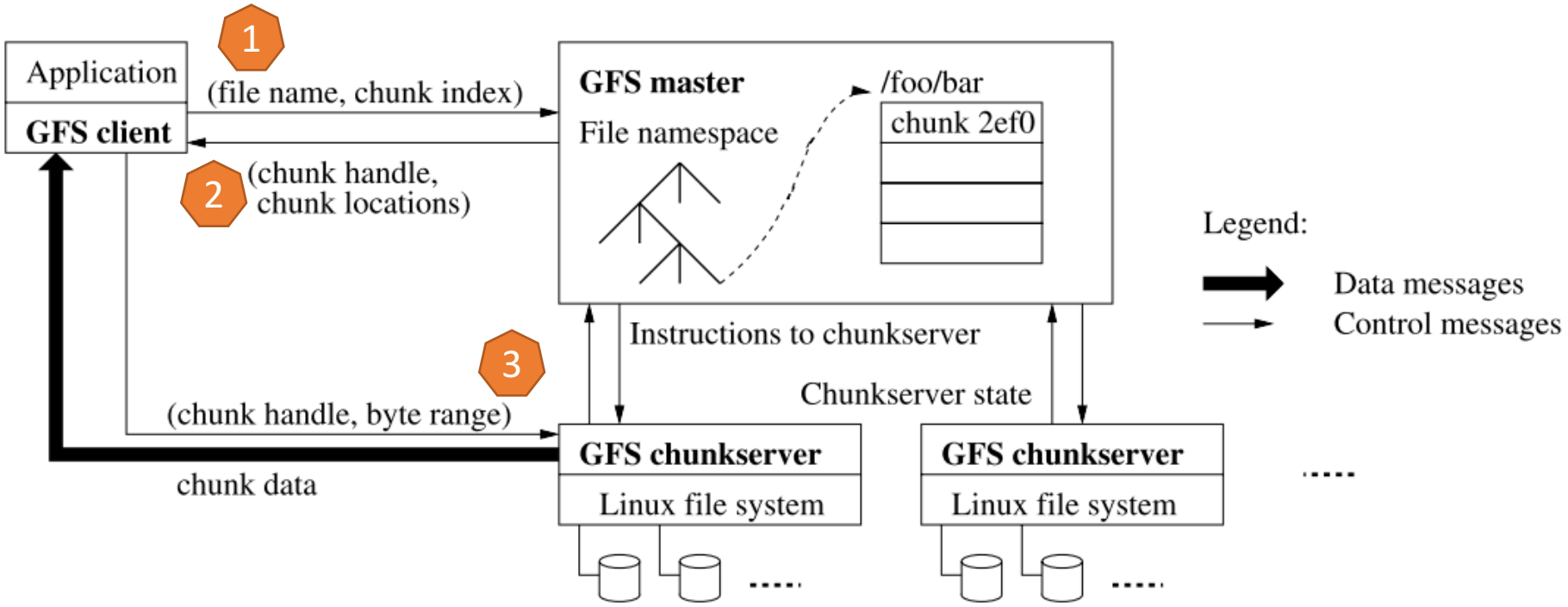
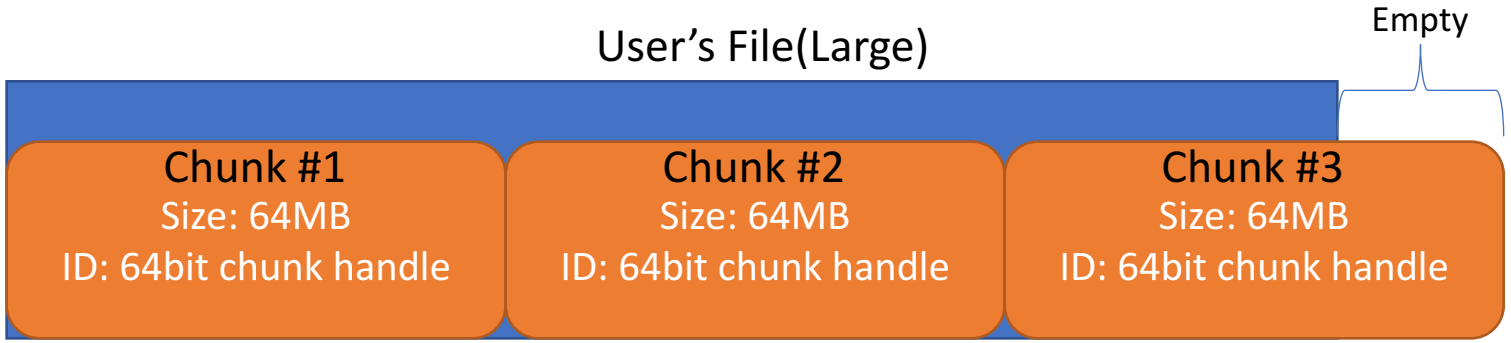
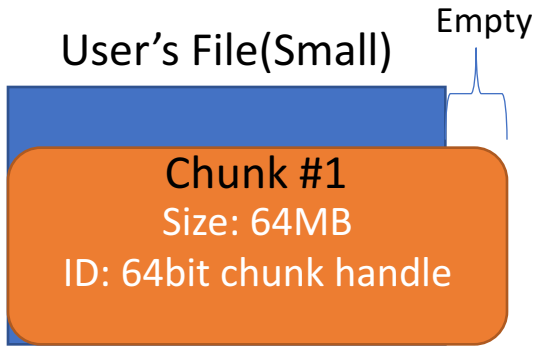


Figure 1: GFS Architecture

Illustration of files and chunks





- Each chunk is hosted by 3(default) different chunk server.

- User needs to do:
  1. Translate file name and offset to file name and chunk index.
  2. Exchange file name and chunk index for chunk handle and chunk location from master
  3. Get File from chunk server with chunk handle and byte range

Single Master

- Advantage:
  - Global knowledge to decide the location of chunks.
- Possible drawback:
  - Bottleneck.
- Solution:
  - Cache (chunk location and chunk handle) on client.
  - Larger chunk size.
    - A chunk may cover more region of a file.

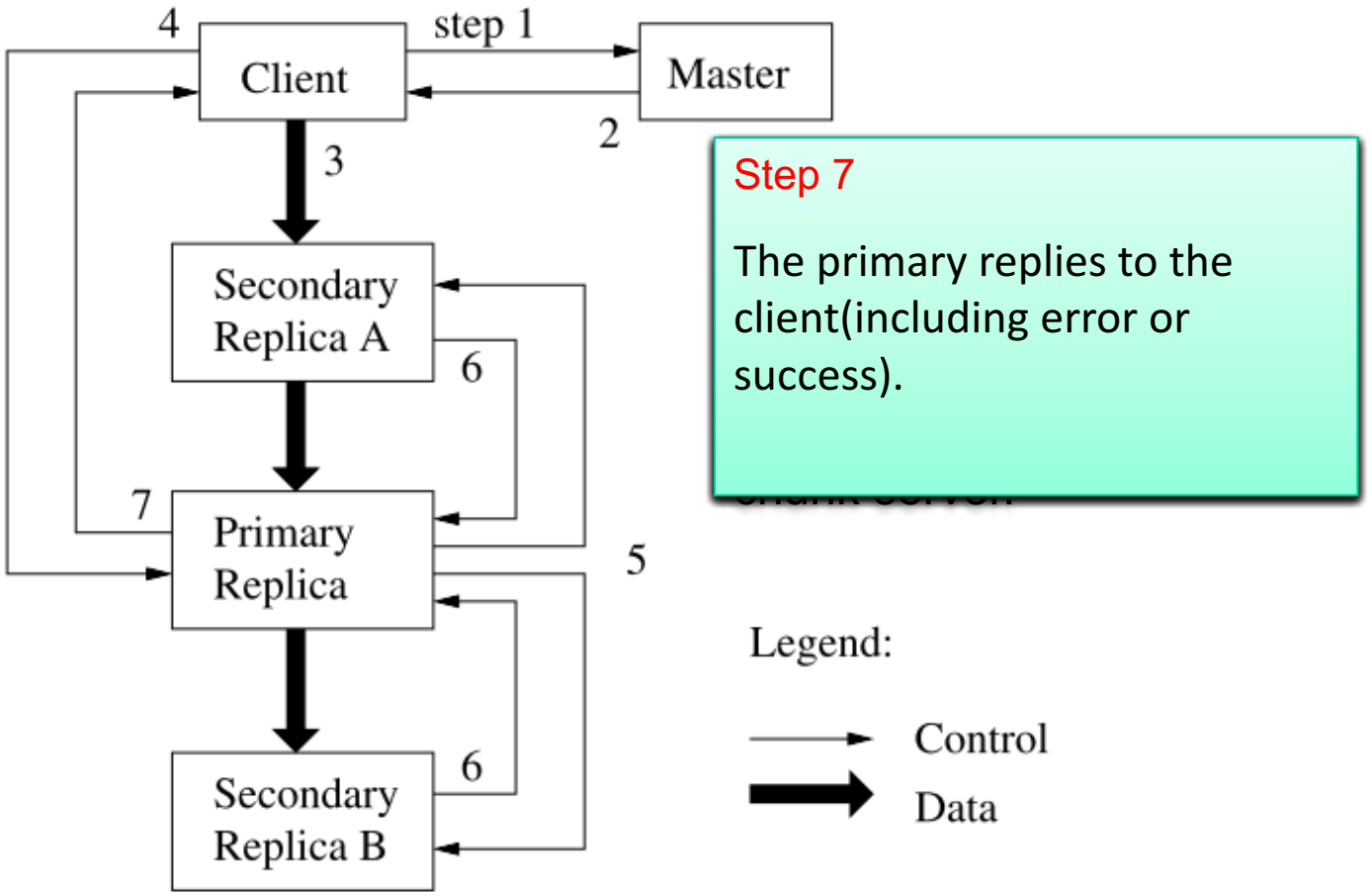
Chunk Size

- Larger size:
  - 64MB (much larger than typical file system block sizes).
  - Chunk is store in chunk server as a plain Linux file.
- Benefits:
  - Reduce masters overhead when clients read or write.
  - It reduces the size of the metadata stored on the master.
    - Metadata(keeps in master server’s memory):
      - The file and chunk namespaces(described later)
      - The mapping from files to chunks
      - Version of chunk

HeartBeat and Operation Log

- HeartBeat messages(periodically)
  - Master controls all chunk placement and monitors chunk server status.
- Operation Log:
  - Contains a historical record of critical metadata changes.
  - Serves as a logical time line that defines the order of concurrent operations.
  - Failover:
    - Checkpoint: replicate the whole metadata in memory to hard disk
    - Store checkpoint data and log both locally and remotely.
    - If fail: Replay the operation log from checkpoint

Write control and data flow



Step 7

The primary replies to the client(including error or success).

Figure 2: Write Control and Data Flow

Data flow

- We decouple the flow of data from the flow of control to use the network efficiently.
  - Control flows: from the client to the primary and then to all secondaries.
  - Data flow: is pushed linearly along a *carefully picked chain* of chunk servers in a *pipelined fashion*.
    - Forwards the data to the "closest" machine in the network topology that has not received it.
  - Ideal time for transferring B bytes to R replicas:
    - $B/T + RL$
    - T is the network throughput and L is latency to transfer bytes between two machine

Namespace Management and Locking

- Allow multiple operations to be active in master by using locking to ensure proper serialization.
  - Recall that GFS does not have a per-directory data structure.
    - It only store file and chunks mapping.
  - So, GFS logically *represents its namespace as a lookup table mapping full pathnames* to metadata.
  - By using read/write lock on namespace tree to ensure serialization.

Data Integrity

- A chunk is broken up into 64 KB blocks and each has a corresponding 32 bit checksum.
  - This checksum is stored in chunk server's memory.
- For reads, *the chunk server verifies the checksum of data blocks* that overlap the read range *before returning any data.*
  - Low overhead:
    - Checksum calculation can often be overlapped with I/Os
- During idle periods, chunk servers can scan and verify the contents of inactive chunks

Conclusion

- GFS demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware.
  - Treat component failures as the norm rather than the exception.
  - Optimize for huge files that are mostly appended to (perhaps concurrently).
- GFS provides fault tolerance by
  - Constant monitoring, replicating crucial data, and fast and automatic recovery.
- High aggregate throughput to many concurrent readers and writers performing a variety of tasks.