



東南大學
SOUTHEAST UNIVERSITY

OPERATING SYSTEM CONCEPTS

.....

Chapter 12. File-System Implementation

A/Prof. Kai Dong



Warm-up

File System Measurement Summary

Most files are small

Average file size is growing

Most bytes are stored in large files

File systems contains lots of files

File systems are roughly half full

Directories are typically small

Roughly 2K is the most common size

Almost 200K is the average

A few big files use most of the space

Almost 100K on average

Even as disks grow, file systems remain ~50% full

Many have few entries; most have 20 or fewer

Objectives



- To describe the details of implementing local file systems and directory structures.
- To describe the implementation of remote file systems.
- To discuss block allocation and free-block algorithms and trade-offs.



Contents

1. Typical File System
2. Fast File System
3. FSCK and Journaling
4. Log-Structured File System
5. Network File System
6. Andrew File System



Contents

1. Typical File System
2. Fast File System
3. FSCK and Journaling
4. Log-Structured File System
5. Network File System
6. Andrew File System



Typical File System

Very Simple File System

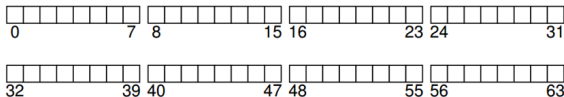
- We are now discussing a simple file system implementation, known as **VSFS** (the **Very Simple File System**)
 - A simplified version of a typical UNIX file system.
- You should understand
 - **Data structures:** what types of on-disk structures are utilized by the file system to organize its data and metadata?
 - **Access methods:** How does it map the calls made by a process onto its structures?



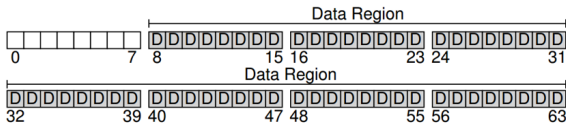
Typical File System

Overall Organization

- Divide the disk into blocks (with a commonly-used block size of 4 KB).
 - Assume a really small disk, with just 64 blocks.



- Reserve a fixed portion of the disk for the data region
 - Say the last 56 of 64 blocks on the disk:

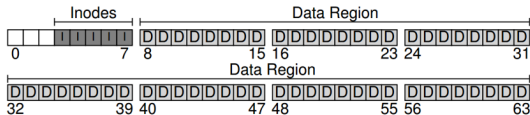




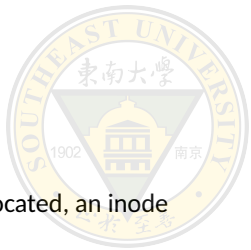
Typical File System

Overall Organization (contd.)

- To track information about each file, the **inodes** are stored in the **inode table**.
 - Assume 5 of 64 blocks for inodes.



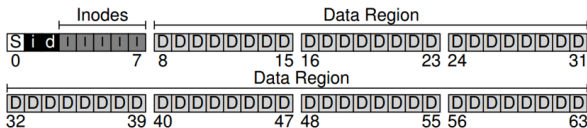
- Assuming 256 bytes per inode, our file system contains ? total inodes.
- This number represents the maximum number of files we can have in our file system.
- How could the file system know which inode /data block is free?



Typical File System

Overall Organization (contd.)

- To track whether inodes or data blocks are free or allocated, an inode bitmap and a data bitmap are required.



- Optional data structures include a free list.
- The remaining block is the superblock.
 - It contains information about this particular file system, including, for example, how many inodes and data blocks are in the file system, where the inode table begins, and so forth.
 - When mounting a file system, the operating system will read the superblock first.



Typical File System

File Organization: the Inode

- An inode in UNIX FS is a **File Control Block** (FCB).
- The name **inode** is short for **index node**.
- Each inode is implicitly referred to by a number, called the **inumber**.

The Inode Table (Closeup)

				iblock 0				iblock 1				iblock 2				iblock 3				iblock 4															
Super	i-bmap	d-bmap		0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67												
				4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71												
				8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75												
				12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79												
0KB				4KB				8KB				12KB				16KB				20KB				24KB				28KB				32KB			

- How to read a given inode number X ?
- By calculating the offset into the inode region ($X \cdot \text{sizeof}(\text{inode})$), add it to the start address of the inode table on disk.



Typical File System

File Organization: the Inode (contd.)

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Simplified Ext2 Inode



Typical File System

File Organization: the Inode (contd.)

- How an inode refers to where data blocks are.
- Suppose a simple approach:
 - To have one or more **direct pointers** (disk addresses) inside the inode; each pointer refers to one disk block that belongs to the file.
- Such an approach is limited:
 - For example, if you want to have a file that is really big (e.g., bigger than the size of a block multiplied by the number of direct pointers).
- Solution: To make use of **indirect pointers**.
 - Instead of pointing to a block that contains user data, it points to a block that contains more pointers, each of which point to user data.
 - » What is the maximum size, with 12 direct pointers and 1 indirect pointer



Typical File System

The Multi-Level Index

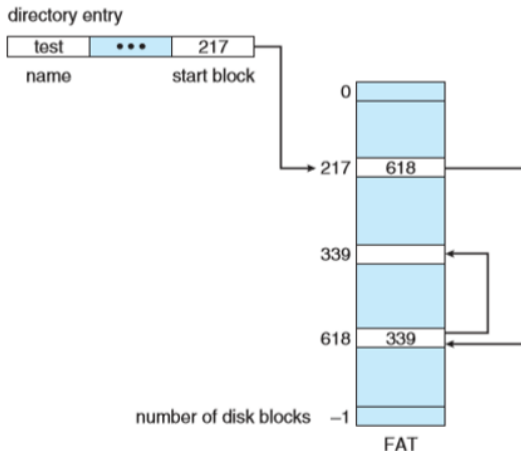
- To support even larger files, by adding a
 - **double indirect pointer**
 - » What is the maximum size, with 12 direct pointers, 1 indirect pointer, 1 double indirect pointer
 - **triple indirect pointer**
 - » What is the maximum size, with 12 direct pointers, 1 indirect pointer, 1 double indirect pointer, 1 triple indirect pointer
- Or, by using **extents** instead of pointers. (ext4)
 - An extent is simply a disk pointer plus a length (in blocks) to specify the on-disk location of a file.
 - Extent-based file systems often allow for more than one extent.
 - Less flexible but more compact.



Typical File System

FAT

- FAT (File-Allocation Table)





Typical File System

NTFS

- NTFS (new technology file system) with MFT (master file table)
- A record in MFT (each is 1KB in size):
 - Body contains attr. data, or a pointer to an extent

Head (ASCII FILE + ...)			
Attr.	Attr. 1	Head	0x10
		Body	Standard Information
	Attr. 2	Head	0x30
		Body	File Name
	...		
	Attr. 8	Head	0x80
		Body	Data
	...		
FF FF FF FF			



Typical File System

Allocation Method

- **Contiguous allocation**
 - Each file occupies a set of contiguous blocks on the disk.
 - ext4, ntfs
- **Linked allocation**
 - Each file is a linked list of disk blocks
 - fat
- **Indexed allocation**
 - Brings all pointers together into the index block
 - ext2, ext3



Typical File System

Directory Organization

- A directory basically contains a list of (entry name, inode number) pairs.

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

- Deleting a file (e.g., calling *unlink()*) can leave an empty space in the middle of the directory, and hence there should be some way to mark that as well (e.g., with a reserved inode number such as zero).
- Such a delete is one reason the record length (reclen) is used: a new entry may reuse an old, bigger entry and thus have extra space within.
- A directory has an inode, somewhere in the inode table (with the type field of the inode marked as “directory” instead of “regular file”).



Typical File System

Access Paths: Reading and Writing

- Suppose we want to open a file, e.g., /foo/bar, read it, then close it.
- Opening a file from disk

`open("/foo/bar", O_RDONLY)`

- The file system must traverse the pathname and thus locate the desired inode.
 1. Read the inode of the root directory which is simply called /.
 2. Look inside the inode to find pointers to data blocks, which contain the contents of the root directory.
 3. Find the entry for foo, and the inode number.
 4. ...



Typical File System

Access Paths: Reading and Writing (contd.)

- Reading a file from disk

read()

1. The first read (at offset 0 unless *lseek()* has been called) will thus read in the first block of the file, consulting the inode to find the location of such a block.
2. ...



Typical File System

Access Paths: Reading and Writing (contd.)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
<i>open(bar)</i>			read		read	read				
				read			read			
<i>read()</i>					read			read		
					write					
<i>read()</i>					read				read	
					write					
<i>read()</i>					read					read
					write					



Typical File System

Access Paths: Reading and Writing (contd.)

- Writing to disk is a similar process

write()

- Unlike reading, writing to the file may also allocate a block.
- Each write to a file logically generates five I/Os:
 - one to read the data bitmap (which is then updated to mark the newly-allocated block as used),
 - one to write the bitmap (to reflect its new state to disk),
 - two more to read and then write the inode (which is updated with the new block's location), and
 - finally one to write the actual block itself.



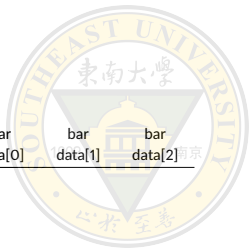
Typical File System

Access Paths: Reading and Writing (contd.)

- Writing to disk

`write()`

- Considering file creation, the total amount of I/O traffic to do so is quite high:
 - one read to the inode bitmap (to find a free inode),
 - one write to the inode bitmap (to mark it allocated),
 - one write to the new inode itself (to initialize it),
 - one write to the data of the directory (to link the high-level name of the file to its inode number), and
 - one read and write to the directory inode to update it.
 - if the directory needs to grow to accommodate the new entry, additional I/Os (i.e., to the data bitmap, and the new directory block) will be needed too.



Typical File System

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
<i>create</i> (/foo/bar)		read write	read	read		read	read write			
<i>write()</i>	read write				read			write		
<i>write()</i>	read write				read				write	
<i>write()</i>	read write				read					
<i>write()</i>	read write				read					



Contents

1. Typical File System
- 2. Fast File System**
3. FSCK and Journaling
4. Log-Structured File System
5. Network File System
6. Andrew File System



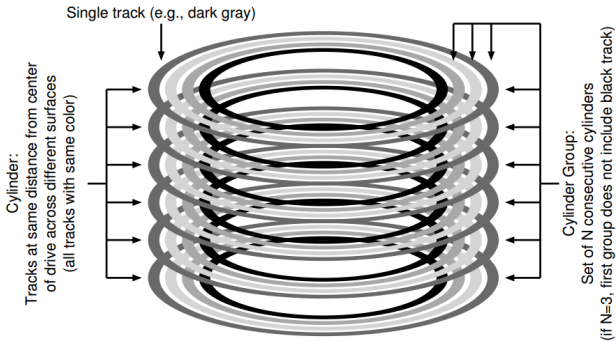
Fast File System

- Performance problems
 - Expensive positioning costs, if data was spread all over the place
 - » The data blocks of a file were often very far away from its inode, thus inducing an expensive seek whenever one first read the inode and then the data blocks.
 - Fragmented file system, if the free space was not carefully managed.
 - The original block size was too small (512bytes).
- Fast File System (FFS): disk awareness
 - Design the file system structures and allocation policies to be “disk aware” and thus improve performance.
 - It keeps the same interface to the FS, but changes the internal implementation.

Fast File System



- Organizing structure: the cylinder group





Fast File System

- Modern drives do not export enough information for the file system to truly understand whether a particular cylinder is in use.
- Modern file systems (such as Linux ext2, ext3, and ext4) instead organize the drive into block groups.
- Whether you call them cylinder groups or block groups, these groups are the central mechanism that FFS uses to improve performance.
 - By placing two files within the same group, FFS can ensure that accessing one after the other will not result in long seeks across the disk.



Fast File System

- FFS keeps within a single cylinder group all the structures you might expect a file system to have.



- FFS keeps a copy of the **super block** (S) in each group for reliability reasons.
- A per-group **inode bitmap** (ib) and **data bitmap** (db) to track whether the inodes and data blocks of the group are allocated.
- The **inode** and **data block** regions are just like those in the previous very-simple file system.



Fast File System

- Policies: how to allocate files and directories
- The basic principle is: keep related stuff together (and its corollary, keep unrelated stuff far apart).
- Placement heuristics: e.g.,
 - For directories
 - » Place a directory in the cylinder group with a low number of allocated directories and a high number of free inodes.
 - For files
 - » First, it makes sure (in the general case) to allocate the data blocks of a file in the same group as its inode.
 - » Second, it places all files that are in the same directory in the cylinder group of the directory they are in.



Fast File System

- If a user creates four files, /a/b, /a/c, /a/d, and b/f
- In general FS

group	inodes	data
0	/-----	/-----
1	a-----	a-----
2	b-----	b-----
3	c-----	c-----
4	d-----	d-----
5	e-----	e-----
6	f-----	f-----
7	-----	-----
...		

- In FFS

group	inodes	data
0	/-----	/-----
1	acde-----	acdde-----
2	bf-----	bff-----
3	-----	-----
...		



Fast File System

- The Large-File Exception
 - Without a different rule, a large file would entirely fill the block group it is first placed within (and maybe others).
 - Filling a block group in this manner is undesirable, as it prevents subsequent “related” files from being placed within this block group, and thus may hurt file-access locality.
 - After some number of blocks are allocated into the first block group, FFS places the next “large” chunk of the file in another block group.
 - A tradeoff here, since spreading blocks of a file across the disk will hurt performance.



Fast File System

- If a user creates one big file, /a
- In general FS-- not enough room for new files in the group in the root directory (/).

group	inodes	data			
0	/a-----	/aaaaaaaa	aaaaaaaaaa	aaaaaaaaaa	a-----
1	-----	-----	-----	-----	-----
...					

- In FFS

group	inodes	data			
0	/a-----	/aaaaa----	-----	-----	-----
1	-----	aaaaa----	-----	-----	-----
2	-----	aaaaa----	-----	-----	-----
3	-----	aaaaa----	-----	-----	-----
4	-----	aaaaa----	-----	-----	-----
5	-----	aaaaa----	-----	-----	-----
6	-----	-----	-----	-----	-----
...					



Contents

1. Typical File System
2. Fast File System
3. FSCK and Journaling
4. Log-Structured File System
5. Network File System
6. Andrew File System



FSCK and Journaling

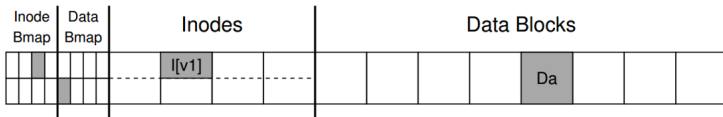
- Considering consistency
 - How to update persistent data structures despite the presence of a power loss or system crash?
- Crash-consistency problem
 - Imagine you have to update two on-disk structures, A and B, in order to complete a particular operation. Because the disk only services a single request at a time, one of these requests will reach the disk first (either A or B). If the system crashes or loses power after one write completes, the on-disk structure will be left in an inconsistent state.
- How to update the disk despite crashes?
- Two approaches
 - A **file system checker** (fsck)
 - **Journaling** (also known as write-ahead logging)



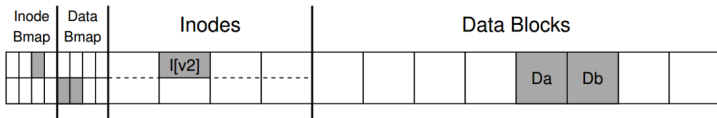
FSCK and Journaling

A Detailed Example

- From



- To



- Three separate writes to the disk are required



FSCK and Journaling

A Detailed Example (contd.)

- Crash Scenarios: Imagine only a single write succeeds.
 - Just the data block (Db) is written to disk.
 - » This case is not a problem at all, from the perspective of file-system crash consistency.
 - Just the updated inode (I[v2]) is written to disk.
 - » If we trust the inode pointer, we will read garbage data from the disk.
 - » File-system inconsistency: The on-disk bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has.
 - Just the updated bitmap (B[v2]) is written to disk.
 - » File-system inconsistency: The bitmap indicates that block 5 is allocated, but there is no inode that points to it.
 - » Space leak: block 5 would never be used by the file system.



FSCK and Journaling

A Detailed Example (contd.)

- Crash Scenarios (contd.): Two writes succeeds.
 - The inode ($I[v2]$) and bitmap ($B[v2]$) are written to disk, but not data (Db).
 - » The file system metadata is completely consistent.
 - » But block 5 has garbage in it.
 - The inode ($I[v2]$) and the data block (Db) are written, but not the bitmap ($B[v2]$).
 - » File-system inconsistency.
 - The bitmap ($B[v2]$) and data block (Db) are written, but not the inode ($I[v2]$).
 - » File-system inconsistency.
 - » We have no idea which file block 5 belongs to.



FSCK and Journaling

Solution #1: FSCK

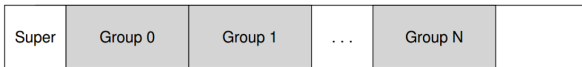
- **Superblock:** If fsck finds a suspect (corrupt) superblock; in this case, the system (or administrator) may decide to use an alternate copy of the superblock.
- **Free blocks:** Fsck scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of which blocks are currently allocated within the file system. It uses this knowledge to produce a correct version of the allocation bitmaps; thus, if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes.
- **Inode state:** Each inode is checked for corruption or other problems. Suspect inode is cleared by fsck.
- **Inode links:** Fsck scans through the entire directory tree to verify the link count of each allocated inode.
- **Duplicates:** Fsck checks for duplicate pointers. If one inode is obviously bad, it may be cleared. Alternately, the pointed-to block could be copied, thus giving each inode its own copy as desired.
- **Bad blocks:** A pointer is considered “bad” if it obviously points to something outside its valid range.
- **Directory checks:** Fsck performs additional integrity checks on the contents of each directory, e.g., making sure that each inode referred to in a directory entry is allocated and no directory is linked to more than once.



FSCK and Journaling

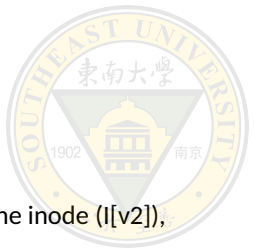
Solution #2: Journaling

- Fscck is too slow and it allows inconsistencies happen and then find and fix them later when rebooting.
- An alternate solution is **journaling** (a.k.a. write-ahead logging).
 - When updating the disk, before overwriting the structures in place, first write down a little note (somewhere else on the disk, in a well-known location) describing what you are about to do.
 - If a crash takes places, the note tells exactly what to fix (and how to fix it) after a crash, instead of having to scan the entire disk.
- Linux ext2 without journaling



- Linux ext3 with journaling

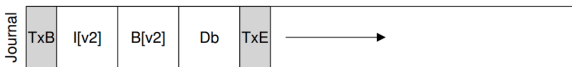




FSCK and Journaling

Solution #2: Journaling (contd.)

- Recall our update example, where we wish to write the inode ($I[v2]$), bitmap ($B[v2]$), and data block (Db) to disk.
- Before any writing, we are now first going to write them to the log (a.k.a. journal).



- The transaction begin (TxB) tells us about this update, and contains a transaction identifier (TID).
- The middle three blocks just contain the exact contents of the blocks themselves. (physical logging vs. logical logging).
- The final block (TxE) is a marker of the end of this transaction, and also contains the TID.



FSCK and Journaling

Solution #2: Journaling (contd.)

- Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system; this process is called **checkpointing**.
- To checkpoint the file system (i.e., bring it up to date with the pending update in the journal), we issue the writes $I[v2]$, $B[v2]$, and Db to their disk locations.
- If these writes complete successfully, we have successfully checkpointed the file system and are basically done.



FSCK and Journaling

Solution #2: Journaling (contd.)

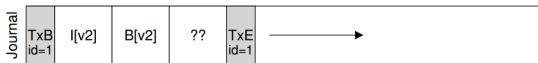
- Protocol to update the file system
 1. **Journal write**: Write the transaction, including a transaction-begin block, all pending data and metadata updates, and a transaction-end block, to the log; wait for these writes to complete.
 2. **Checkpoint**: Write the pending metadata and data updates to their final locations in the file system.



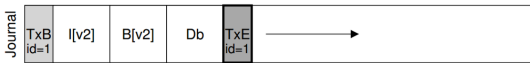
FSCK and Journaling

Solution #2: Journaling (contd.)

- When a crash occurs during the writes to the journal.
 - Issue each write at a time, waiting for each to complete, and then issuing the next. But this is slow.
 - Issue all five block writes at once, as this would turn five writes into a single sequential write and thus be faster. However this is unsafe.



- Thus the file system issues the transactional write in two steps.





FSCK and Journaling

Solution #2: Journaling (contd.)

- Protocol to update the file system
 1. **Journal write**: Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete.
 2. **Journal commit**: Write the transaction commit block (containing TxE) to the log; wait for write to complete; transaction is said to be committed.
 3. **Checkpoint**: Write the pending metadata and data updates to their final locations in the file system.



FSCK and Journaling

Solution #2: Journaling (contd.)

- A crash may happen at any time during this sequence of updates.
 - If the crash happens before the transaction is written safely to the log (i.e., before Step 2 above completes), then our job is easy: the pending update is simply skipped.
 - If the crash happens after the transaction has committed to the log, but before the checkpoint is complete, the file system can recover the update.



FSCK and Journaling

Solution #2: Journaling (contd.)

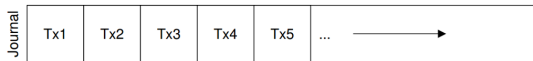
- Batching Log Updates
- Now suppose we create multiple files in a row in the same directory.
 - To create one file, one has to update a number of on-disk structures, minimally including:
 - » the inode bitmap,
 - » the newly-created inode of the file,
 - » the data block of the parent directory containing the new directory entry,
 - » the parent directory inode,
 - » etc.
 - We are writing these same blocks over and over.
- Solution: buffer all updates into a global transaction.



FSCK and Journaling

Solution #2: Journaling (contd.)

- Making The Log Finite
 - The log is of a finite size. If we keep adding transactions to it (as in this figure), it will soon fill.



- Use a **journal superblock** to record enough information to know which transactions have not yet been checkpointed, and thus reduces recovery time as well as enables re-use of the log in a circular fashion.



FSCK and Journaling

Solution #2: Journaling (contd.)

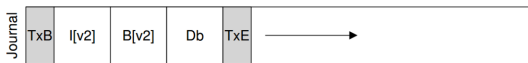
- Protocol to update the file system
 1. **Journal write**: Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete.
 2. **Journal commit**: Write the transaction commit block (containing TxE) to the log; wait for write to complete; transaction is said to be committed.
 3. **Checkpoint**: Write the pending metadata and data updates to their final locations in the file system.
 4. **Free**: Some time later, mark the transaction free in the journal by updating the journal superblock.



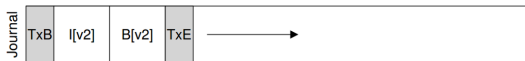
FSCK and Journaling

Solution #2: Journaling (contd.)

- Journaling Mode: Data / Ordered / Unordered.
 - Data journaling (as in Linux ext3) requires writing data twice to the disk.



- Metadata journaling, does not write data to the journal.



- » When should we write Db?
- » Ordered / Unordered



FSCK and Journaling

Solution #2: Journaling (contd.)

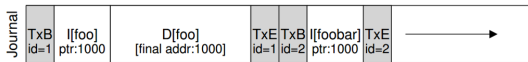
- Protocol to update the file system
 1. **Data write**: Write data to final location; wait for completion (the wait is optional).
 2. **Journal metadata write**: Write the begin block and metadata to the log; wait for writes to complete.
 3. **Journal commit**: Write the transaction commit block (containing TxE) to the log; wait for write to complete; transaction is said to be committed.
 4. **Checkpoint**: Write the pending metadata and data updates to their final locations in the file system.
 5. **Free**: Some time later, mark the transaction free in the journal by updating the journal superblock.



FSCK and Journaling

Solution #2: Journaling (contd.)

- Block Reuse
 - User adds an entry to the directory *foo*, assume the location of the *foo* directory data is block 1000.
 - User deletes everything in the directory as well as the directory itself, freeing up block 1000 for reuse.
 - User creates a new file *foobar*, which ends up reusing the same block 1000.



- Now assume a crash occurs and all of this information is still in the log. What will happen?



FSCK and Journaling

Solution #2: Journaling (contd.)

- Solutions:
 - Never reuse blocks until the delete of said blocks is checkpointed out of the journal
 - Linux ext3: add a new type of record (**revoke**) between two transactions. Such revoked data is never replayed.



FSCK and Journaling

Solution #3: Copy-on-Write

- Other Approaches include **copy-on-write** (in storage not memory management).
 - Never overwrites files or directories in place; rather, it places new updates to previously unused locations on disk.
 - Doing so makes keeping the file system consistent straightforward.
 - We will discuss the **log-structured file system** (LFS), which is an early example of a COW.



Contents

1. Typical File System
2. Fast File System
3. FSCK and Journaling
- 4. Log-Structured File System**
5. Network File System
6. Andrew File System



Log-Structured File System

- Observations:
 - System memories are growing
 - » As more data is cached in memory, disk traffic increasingly consists of writes, as reads are serviced by the cache
 - There is a large gap between random I/O performance and sequential I/O performance
 - » Hard-drive transfer bandwidth has increased a great deal over the years, however seek and rotational delay costs have decreased slowly
 - Existing file systems perform poorly on many common workloads
 - » Even FFS incurs many short seeks and subsequent rotational delays
 - File systems are not RAID-aware
 - » Small writes problem in RAID-4 and RAID-5



Log-Structured File System

- Log-structured File System (LFS)
 - 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 is used efficiently, and performance of the file system approaches its zenith.
 - How can a file system transform all writes into sequential writes?



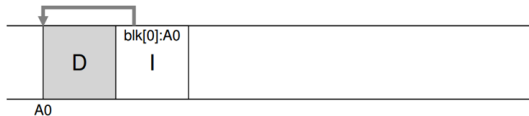
Log-Structured File System

- Writing To Disk Sequentially

- Imagine we are writing a data block D to a file, at disk address A0.



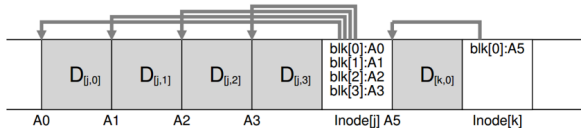
- When a user writes a data block, it is not only data that gets written to disk; there is also other metadata that needs to be updated.





Log-Structured File System

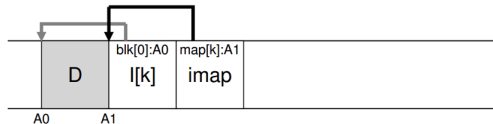
- With write buffering



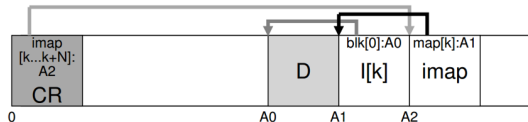
- How much to buffer?
- Some calculation:
 - With a disk with a positioning time of 10milliseconds and peak transfer rate of 100MB/s; assume we want an effective bandwidth of 90% of peak.
 - In this case, $D = 9MB$.

Log-Structured File System

- How to find inodes?
 - LFS uses a level of indirection between inode numbers and the inodes through a data structure called the inode map (imap).



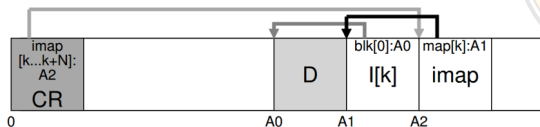
- How to find imaps?
 - The file system must have some fixed and known location to begin a file lookup. Such location is the checkpoint region (CR).



- Note the checkpoint region is only updated periodically (say every 30 seconds or so), and thus performance is not ill-affected.



Log-Structured File System

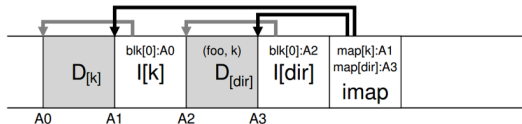


- Reading a file
 1. Read the checkpoint region, which contains pointers to the entire inode map.
 2. Read the inode map, and caches it in memory.
 3. Looks up the inode-number to inode-disk-address mapping in the imap.
 4. Reads in the most recent version of the inode.
 5. ... (Similar as typical UNIX file system).



Log-Structured File System

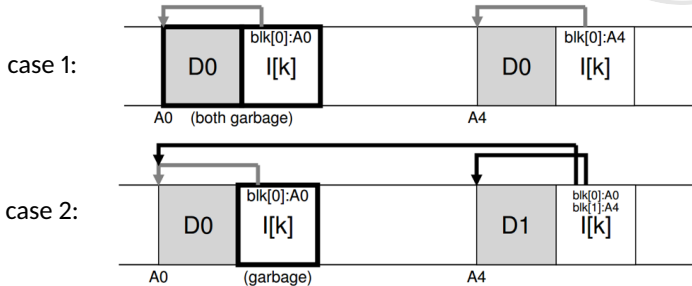
- Considering directories
 - When creating a file on disk, LFS must both write a new inode, some data, as well as the directory data and its inode that refer to this file.



- Recursive update problem (when updating an inode)
- Solution is simple: Even though the location of an inode may change, the change is never reflected in the directory itself. (This is because $imap$ instead of directory, now record the location of inode.)

Log-Structured File System

- Garbage Collection

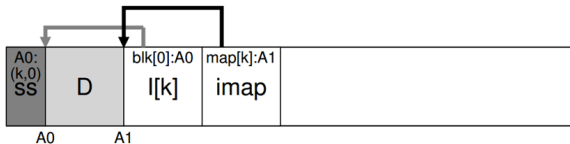


- Versioning file system keeps track of the different versions.
- Using compaction, to get rid of free holes.



Log-Structured File System

- Determining Block Liveness
 - LFS includes, for each data block D, its inode number (which file it belongs to) and its offset (which block of the file this is). This information is recorded in a structure at the head of the segment known as the segment summary block.





Log-Structured File System

- Crash Recovery And The Log
 - What happens if the system crashes while LFS is writing to disk?
 - » LFS organizes the writes in a log pointed by the checkpoint region. (the checkpoint region points to a head and tail segment, and each segment points to the next segment to be written.)
 - » To ensure that the CR update happens atomically, LFS actually keeps two CRs.
 - » Suppose a crash happens during writing to a segment. Roll forward.



Contents

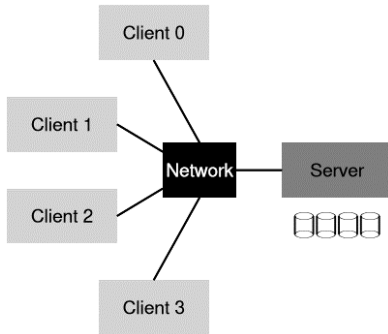
1. Typical File System
2. Fast File System
3. FSCK and Journaling
4. Log-Structured File System
- 5. Network File System**
6. Andrew File System



Network File System

- In such an environment, there are a number of client machines and one server (or a few); the server stores the data on its disks, and clients request data through well-formed protocol messages.

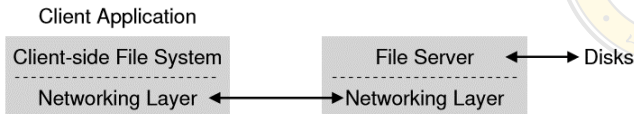
- Benefits
 - Sharing of data
 - Centralized administration
 - Security





Network File System

A Basic Distributed File System (contd.)



- On the client side, there are client applications which access files and directories through the **client side file system**.
- A client application issues **system calls** to the client-side file system in order to access files which are stored on the server.
- The client-side file system may send a message to the **server-side file system** (or, as it is commonly called, the **file server**) to perform the IO operations.
- A subsequent *read()* of the same block on the client may be **cached** in client memory or on the client's disk even.
- A challenge deals with fast server **crash recovery**.



Network File System

Fast Crash Recovery: Statelessness

- Why not **stateful** protocol?
- Stateful protocol: The server keeps track of everything about what is happening at each client
 - Complicates server crash recovery
 - Has to deal with client crashes

```
1 char buffer[MAX];  
2 int fd = open("foo", O_RDONLY); // get descriptor "fd"  
3 read(fd, buffer, MAX);          // read MAX bytes from foo (via fd)  
4 ...  
5 read(fd, buffer, MAX);          // read MAX bytes from foo (via fd)  
6 close(fd);                      // close file
```



Network File System

Fast Crash Recovery: Statelessness

- Why **stateless** protocol:
- Stateless protocol: The server, by design, does not keep track of anything about what is happening at each client.
 - Each client operation contains all the information needed to complete the request.
 - No fancy crash recovery is needed; the server just starts running again, and a client, at worst, might have to retry a request.
 - Use file handle instead of file descriptor.
 - 3 important components: a volume identifier, an inode number, and a generation number



Contents

1. Typical File System
2. Fast File System
3. FSCK and Journaling
4. Log-Structured File System
5. Network File System
- 6. Andrew File System**



Andrew File System

- Scale:
 - How can one design a distributed file system such that a server can support as many clients as possible?
 - The whole-file caching on the local disk of the client machine that is accessing a file.



Andrew File System

AFS v1

AFSv1 Protocol Highlights

TestAuth	Test whether a file has changed (used to validate cached entries)
GetFileStat	Get the stat info for a file
Fetch	Fetch the contents of file
Store	Store this file on the server
SetFileStat	Set the stat info for a file
ListDir	List the contents of a directory

- Problems.
 - Path-traversal costs are too high;
 - The client issues too many TestAuth protocol messages.



Andrew File System

AFS v2

- A file identifier (FID) (similar to the NFS file handle) instead of pathnames to specify which file a client was interested in
- A callback is simply a promise from the server to the client that the server will inform the client when a file that the client is caching has been modified.