CS-323: OS Exercise Week 13 (Fall 2023)

December 13, 2023

Last name, first name: ______________________________

1 File Systems

(a) Give modification timeline of file system metadata and function calls to open file **/foo/bar/os**, read it and append 2 blocks to it. Suppose it has size 12KB and file system block size is 4KB, and each block is read/written in a separate call. Assume the superblock does not need to be modified.

Please note that close() also be there but it does not require any access to the file system metadata. It only updates open files table. Check Table 1:File System Updates Timeline

Name: ____________________________

	data bitmap	inode bitmap	root inode	foo inode	bar inode	os inode	root data	foo data	bar data	os[0] data	os[1] data	os[2] data	os[3] data	os[4] data
open(os)			read											
							read							
				read										
								read						
					read									
									read					
						read								
read()						read								
										read				
						write								
read()						read								
											read			
						write								
read()						read								
												read		
						write								
write()						read								
	read													
	write													
													write	
						write								
write()						read								
	read													
	write													
														write
						write								

Table 1: File System Updates Timeline

2 Crash Consistency

(a) What is the crash-consistency problem?

We expect file systems to be persistent both in the perspective of the data storage and the metadata, even if confronted with system failure and/or power loss. Given that crashes (software bugs, hardware failure etc.) can occur at arbitrary points in time, to ensure the file system keeps the on-disk image in a reasonable state or to recover if it doesn't so defines the crash-consistency problem. It is basically keeping file system always in consistent state independent of possible failures. Since it happens due to updates it is also called the consistent-update problem.

(b) Conceptualize all crash scenarios for a single block write for an existing file residing in a simple file system consisting only of inode blocks, data blocks, their corresponding bitmap blocks and a super-block.

Appending a single block to a file in such a file system requires multiple updates which can happen in any order: the data block should be written to the storage media, the corresponding bit in the data bitmap should be updated, and the corresponding inode should be updated. A crash can happen at any time.
When only one such update happens:

- **Only the data block was written to the disk:** No information about this block existing is present in either the inode or the data bitmap. This will result in data loss since in the perspective of the file system no such write ever happened; there is no metadata. There is no inconsistency in the file system perspective.
- **Only the inode is updated:** Then the inode will be pointing to garbage data. There is an inconsistency due to metadata mismatch between the inode and the data bitmap. According

to the inode, there exists a block belonging to a file and this block should have value 1 in its corresponding bitmap position but it does not. This inconsistency should be resolved.

- **Only the data bitmap is updated:** Then there will be a space leak in the filesystem. The block becomes unusable by the file system since it is marked as full and but it will never be freed since there is no inode pointing to it. This is an inconsistency which should be resolved.

When only two such updates happen:

- **The inode and the data bitmap are updated:** Again the inode will be pointing to garbage, but there is no inconsistency in file system perspective.
- **The inode is updated and block is written to disk:** There is an inconsistency due to metadata mismatch between the inode and the data bitmap. According to the inode, there exists a block belonging to a file and this block should have value 1 in its corresponding bitmap position but it does not. Even though data is written such state may cause data loss. This inconsistency should be resolved.
- **Data bitmap is updated and block is written:** Then there is an inconsistency due to metadata mismatch between the inode and the data bitmap. According to the bitmap the block exists but it belongs no file. Even though data is written the file has no access to it. This inconsistency should be resolved.

3 FSCK

(a) What is FSCK? How does it work?

`fsck`, file system checker, is a piece of software designed to resolve file system inconsistencies in UNIX-like systems. It runs before file system is mounted and made available; once it finishes, on-disk file system should be in a consistent state. How it works is described below. In a nutshell, it re-creates entire file system virtually in a tree-like structure from metadata and checks and resolves the inconsistencies along the way.
Some of the things it does:

1. Checks if superblock in a reasonable state. If it is corrupted, use one of the backup copies version (if available)
2. Scans inodes, indirect blocks, double indirect blocks etc and virtually create file system tree. It uses such tree to produce the correct version of bitmap allocation (puts its trust to the inodes in this case).
3. Checks inodes state for file type, file size, link count etc. and corrects them if possible. If there exists a problem it marks inode as *suspect* and clears it later (of course by updating corresponding bitmap too).
4. Checks inode links and reference count. If an allocated inode is found but no directory refers to it then it is moved to **lost+found** directory.
5. Checks for duplicates and their state.
6. Checks for bad blocks by scanning all block pointers(!) for valid range, partition number etc. Removes bad block pointers.
7. Checks directories for "." and ".." entries

(b) What is the downside of FSCK?

`fsck` requires intricate knowledge about file system. It is **too slow** and its performance goes even worse with RAID systems since it's not at all aware of them. Although it has fixing mechanisms for consistency, it is not infinitely clever: e.g. data corruptions are not fixable. (Deleting something without making sure of its usage is never a good idea.)

Name: ___________________________

4 Journaling

(a) What is journaling?

It is a proactive solution to the consistent-update problem aiming to keep the system always in a consistent and **correct** state. It also brings **atomicity** to the file system at the OS level since otherwise atomicity can only be achieved by the storage device. Journaling requires writing changes to the disk twice: first, any updates are written as a transaction/chain of transactions and second, actual updates are written to the correct places on the disk. Transactions are high level information describing which structure of the disk will be updated how. Transactions also include start and end blocks (with timestamps) that pave the way to atomicity.

(b) How does journaling differ from FSCK?

FSCK is a reactive approach which focuses on solving inconsistencies after they occur whereas journaling aims to prevent it. Also, FSCK does not guarantee correctness as opposed to journaling. FSCK does not introduce redundancy or any kind of data replication so cannot recover data losses unless data is already in the disk and referenced in inodes whereas journaling supports data recovery.

(c) Suppose for a file system using journaling for consistency (e.g. ext3), there is only one file updated in which a single block is appended since the last checkpoint. Show the timeline and update order file system as well as the journal for data journaling. Assume the superblock does not need to be modified.

D_x = data blocks owned by the existing file
I_x = the inode block owned by the existing file
D = new block
SxT = start transaction
ExT = end transaction
I_b = the file's inode bitmaps
D_b = the file's data block bitmaps

As you may see in Table 2, data journaling writes the data in the journal entry (which is located on the disk) and its actual position in the disk (two disk writes). Metadata of file system is only updated after the data is written to disk for both cases, which prevents the data loss. (If power goes out before writing the data to disk anyhow then say good bye to the data anyway! There's no magic solution.).

The lines in the table are not just logical but also physical borders which enforces some updates happening before others. Those lines can be implemented in hardware level which ensures preserving write order in the presence of disk schedulers and called **write barriers** (i.e. forcing journal writes to happen before data/metadata writes).

Journal				File System	
StartBlock	Metadata	Data	EndBlock	Metadata	Data
SxT	I_x,D_b	D			
done					
	done				
		done			
			ExT		
			done		
				I_x,D_b	D
					done
				done	

Table 2: Data Journaling Timeline