CS 111 Fall 2009

Lecture 12 – Robustness

*By: Yosia Sugialam and Michael Wirijadinata*

**Review**

1. Performance at low level (e.g. anticipatory scheduling, batching)
2. Data structures on disk (e.g. inodes, directories)
3. Robustness (in the presence of hardware problems)
   * Goal is to survive the common “survivable” problems

**Terminology**

1. Error 🡪 Mistake “in your (designer/builder) head”
2. Fault 🡪 Latent defect in Hardware or Software, which may cause problem
3. Failure 🡪 System behaves incorrectly

We want to make a robust system to tolerate faults, so that the faults don’t cause failure

Let’s take a file system design as an example of how to apply robustness in a system.

**Goals of file system**

1. Durability: File system is OK even if Hardware fails (in limited ways)
2. Atomicity: Changes to file system are either made or not made 🡪 this implies that we don’t have to worry about partial transaction which is often very hard to deal with.
3. Performance: Maximum Throughput and minimum latency.

Let us consider a system call function called rename:

Assumption: small directory (few dirents / directory entry)

rename(“a”,”b”)

The steps that it performs are:

1. Read directory into RAM
   * since small directory🡪assumes 1 block
2. Scan through that block looking for directory named “a”
   * check that nothing is named “b”
3. Zap that byte to make it look like b (‘a’🡪’b’)
4. Write directory out from RAM back to disk

What can go wrong? Let us suppose there is a power outage in between those steps. Since all the steps that we defined above are atomic, even if there is a power outage in between any of the steps the system would still behaves correctly.

Now let’s do a rename on large directory and suppose “b” is already exist. We will use a simple algorithm that basically says: “if “b” found, just delete it”.

rename (“a”,”bb”)

The steps that it performs are:

1. Read all blocks into RAM (they need to be contiguous)
2. Find ‘a’ and ‘bb’
3. Mark ‘bb’ as nonexistent
4. Change “a” to “bb” (in a’s entry 🡪 may need to reorganize)

As we can see, we now have several blocks to write out. If a power outage happens between particular steps above, the system will behave incorrectly.

Let us consider a simpler function: link

link (“d/a”, “e/b”)

The steps that it performs are:

1. Read d’s data to find a’s inode number (I)
2. Read e’s data into RAM to find if there’s no b in it (assume no “e/b”)
3. Add directory entry (“b”🡪 I)
4. Write out e’s data block
5. Read inode number I
6. Add 1 to link count
7. Write Inode number I

Say there is a crash between step 4 and 5.

Then after reboot, if we do $ ls –li d/a e/b, we will get:

27916 1 rw- -r- -r-- d/a

27916 1 rw- -r- -r-- e/b

Where 27916 is the inode number and 1 is the link count. It is obvious that this creates a DISASTER as both “d/a” and “e/b” has the same metadata (inode # , link count, etc).

The solution for this problem is that we reorder the steps. The new order looks like this:

1 🡪 2 🡪 6 🡪 7 🡪 3 🡪 4 🡪 5 instead of 1 🡪 2 🡪 3 🡪 4 🡪 5 🡪 6 🡪 7.

In other words, if we assume step 4 to be Y and step 7 to be X, we can reorder the steps as we want (of course as long as it makes sense) except that X < Y (step 7 has to be done before step 4). Let’s analyze the effect of our reordering in more details. Now, if a crash happens between step 4 and 5 we would have

27916 2 rw- -r- -r- d/a

e/b: no such file

We still have a problem here, in which the “d/a” storage is never reclaimed. This is a Performance Problem and the good news is that this would not create DISASTER and thus can be considered TOLERABLE. Moreover, we can write a file system checking that will look at link counts, etc. If it finds a problem, e.g. a block that nobody points to or erroneous link counts, it can reclaim unused storage.

The Unix system actually has such a file system checking function called fsck. It is a private syscall to kernel and it knows the data structure of the file system. It also does read and write blocks directly. It also needs to insist that the file system is not in use at all by any other applications (exclusive access).

Problems in implementing fsck:

1. We cannot use ordinary system calls such as open, close, rename, link, read, write
2. Needs to access the block in the disk directly without using the file system (reads and writes the block directly)
3. Wants exclusive access to file system

Invariants for file system (a part of any serious file system design):

1. Every block on the file system is used for exactly 1 purpose
   * boot, superblock, bitmap, or inode, etc)
2. All reference blocks are initialized to values appropriate for that block.
   * proper uid, inode number
   * no garbage data
3. All reference blocks are marked USED in the bitmap
   * If you’re using a data block somewhere, the bitmap will flag it as 1 (USED)
4. All unreferenced blocks are marked UNUSED in the bitmap

If we want to make a robust system, we MUST know the invariants of the system. This is a part of any serious system design.

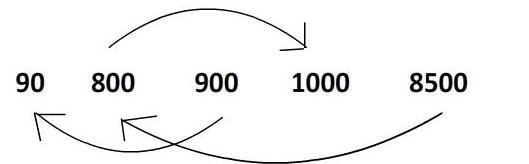
There are consequences for violating the Invariants:

1. Violating invariants no. 1 will cause DISASTER 🡪 file system no longer consistent
2. Violating invariants no. 2 will cause DISASTER also 🡪 garbage inode
3. Violating invariants no. 3 will cause DISASTER also 🡪 processes may overwrite each other’s data
4. Violating invariants no. 4 will cause Memory Leak (Performance Problem)

Therefore, by listing all of the possibility of violations, a system designer can decide which problem can be neglected and tolerated ( in this case invariants no.4).

Next problem: Suppose we already make a robust file system by reordering primitives in file system apps. And suppose we have a disk scheduling that use the elevator algorithm. Also assumes that our X step is located in memory location 900 while the Y step is located in mem. location 90. Ideally, based on our previous solution, we want to execute X first. However, because of the way the elevator algorithm works, the Y step which is located in mem. location 90 will be executed first.

Solution: we can create a modified circular elevator algorithm that works based on dependencies.



Thus the actual order of execution is: 900 8500 90 800 1000

This would solve the problem though doing it this way would sacrifice the efficiency of the algorithm. It is clear that the original elevator algorithm is much faster.

Is there any way to improve the performance of our modified elevator algorithm? Say by decreasing the number of the dependencies?

Yes, we can ignore dependencies involving data block (these are the bulk of the writes) in regular file and keep track of dependencies only for bitmaps, inodes, directories, and superblocks.

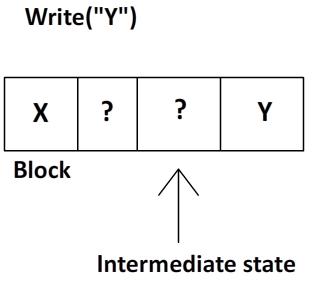
Suppose we want to append to a file:

1. Write new size in inode
2. Write data at the end of file

Suppose a crash occurred in between above steps, what would happen? The file size is not zero, but actually there is no data inside (disk file extended with zeros). But invariants still hold, so do we allow this? The answer depends on the decision of the system designer.

**How to Implement Atomic Write**

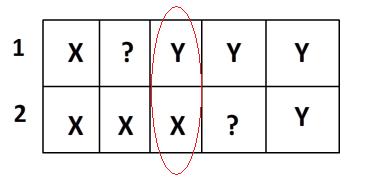
On a non-atomic system:



Contents

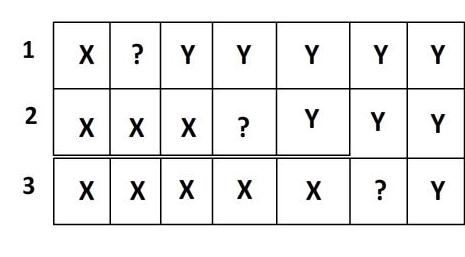
Simple work around:

Trade space for correctness (write multiple copies)



We can recover either x or y every crash. However there is a case where we will be left with both X and Y (marked with circle). In this case, after reboot we don’t know which the correct data is.

Solution: use 3 copies instead of 2.



Write operation: write 1,2,3

Read operation: choose majority, if all 3 disagree choose #1.

Lampson Sturgis assumption:

* Writes can fail, can corrupt nearby storage
* Read can detect bad sector/failed writes
* Sector can go bad spontaneously
* Errors are rare

These days based on those assumptions, people usually implements only 2 or even less copies of the data.