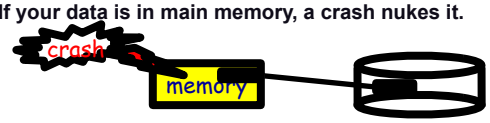
Surviving failure

- OSes and computers crash.
  - **Your file system should not be destroyed.**
- Readings:
  - **Silberschatz:**
    - **7th ed.: ch. 6.9 & 12**
    - **6th ed.: ch 7.9 & 14**
  - **man fsck**

What is the problem?

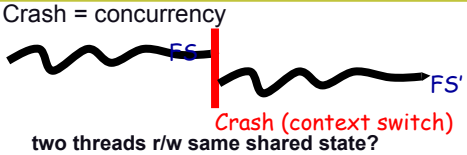
- The Big File System Promise: persistence
  - **It will hold your data until you explicitly delete it**
  - **(and sometimes even beyond that: backup/restore)**
- What's hard about this? Crashes
  - **If your data is in main memory, a crash nukes it.**



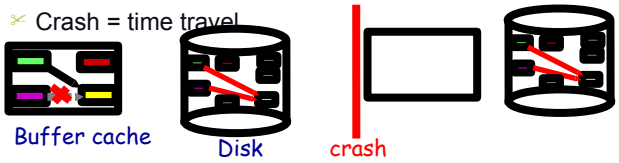
  - **Performance tension: need to cache everything. But if so, then crash = lose everything.**
  - **More fundamental: interesting ops = multiple block modifications, but can only atomically modify disk a sector at a time.**

What failure looks like

- Crash = concurrency



  - **two threads r/w same shared state?**
- Crash = time travel



  - **Current volatile state lost; suddenly go back to old state**
  - **Plus: write back buffer cache = reordered disk writes!**

Fighting failure

- In general, coping with failure consists of first defining a failure model composed of:
  - **Acceptable failures. E.g., the earth is destroyed by the weirdos from Mars. The loss of a file viewed as unavoidable.**
  - **Unacceptable failures. E.g. power outage: lost file not OK**
- Second, devise a recovery procedure for each unacceptable failure:
  - **Takes system from a precisely understood but incorrect state to a new precisely understood and correct state.**
- Dealing with failure is *hard*
  - **Containing effects of failure is complicated.**
  - **How to anticipate everything you haven't anticipated?**

FS Caches: Three main approaches

- Sol'n 1: Throw everything away and start over.
  - **Done for most things (e.g., interrupted compiles).**
  - **Probably not what you want to happen to your email**
- Sol'n 2: Make updates seem indivisible (atomic)
  - **Build arbitrary sized atomic units from smaller atomic ones (e.g., a sector write).**
  - **Similar to how we built critical sections from locks, and locks from atomic instructions.**
- Sol'n 3: Reconstruction
  - **Try to fix things after crash (many Fses do this: "fsck")**
  - **Usually do changes in stylized way so that if crash happens, can look at entire state and figure out where you left off.**

Arbitrary-sized atomic disk operations

- Atomic operation: bundles a set of operations such that they appear to execute indivisibly.
- For disk: construct a pair of operations:
  - **put(blk, address) : writes data in blk on disk at address.**
  - **get(address) -> blk : returns blk at given disk address.**
  - **Such that "put" appears to place data on disk in its entirety or not at all and "get" returns the latest version.**
  - **What we have to guard against: a system crash during a call to "put", which results in a partial write.**
- How? State duplication.
  - **The algorithm was first described in 1961 for the SABRE American Airlines seat-reservation system.**
  - **Still relevant: LFS uses a variant to write checkpoints**

SABRE atomic disk operations

```
void atomic-put(data)
      version++; #unique int
      put(version, V1);
      put(data, D1);
      put(version, V2);
      put(data, D2);
```

```
blk atomic-get()
      V1data := get(V1)
D1data := get(D1);
      V2data := get(V2);
D2data := get(D2);
      if(V1data == V2data)
      return D1data;
      else
      return D2data;
```

- V1, D1, V2, D2: (different) disk addresses
- version is a integer in volatile storage
- a call to atomic-put("seat 25") might result in:
  { #2, "seat 25", #2, "seat 25" }

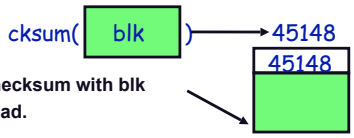
Does it work?

- Assume we have correctly written to disk:
  { #2, "seat 25", #2, "seat 25" }
- And that the system has crashed during the operation atomic-put ("seat 31")
- There are 6 cases, depending on where we failed in atomic-put:

put # fails	possible disk contents	atomic-get returns?
Before	{#2, "seat 25", #2, "seat 25"}	
the first	{#2.5, "seat 25", #2, "seat 25" }	
the second	{#3, "seat 35", #2, "seat 25"}	
the third	{#3, "seat 31", #2.5, "seat 25"}	
the fourth	{#3, "seat 31", #3, "seat 35"}	
After	{#3, "seat 31", #3, "seat 31"}	

Two assumptions

- Once data written, the disk returns it correctly
  If data can be corrupted? Detect using checksums.
  Checksum ~ a hash function s.t. corrupted block gives different value.



  detection: store checksum with blk and re-check on read.
- Disk is in a correct state when atomic-put starts
  before doing the next "put" after a failure, we need to repair the disk to get it back to a correct state
  tricky part: if we crash during recovery, the disk should not get even more trashed!

Recovery: built on idempotent operations

```
void recover(void) {
  V1data := get(V1);  # following 4 ops same as in a-get
  D1data := get(D1);
  V2data := get(V2);
  D2data := get(D2);
  if (V1data == V2data)
      if(D1data != D2data)
          # if we crash & corrupt D2, will get here again.
          put(D1data, D2);
  else
      # if we crash and corrupt D1 will get back here
      put(D2data, D1);
      # if we crash and corrupt V1, will get back here
      put(V2data, V1);
  version := V1data
```

The power of state duplication

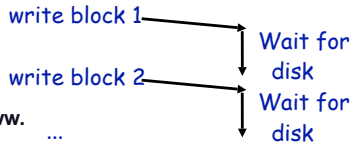
- Most approaches to tolerating failure have at their core a similar notion of state duplication
  Want a reliable tire? Have a spare.
  Want a reliable disk? Keep a tape backup. If disk fails, get data from backup. (Make sure not in same building.)
  Want a reliable server? Have two, with identical copies of the same information. Primary fails? Switch. (Make sure not on same side of the country)
  Like caches (another state duplication): easy to generalize to more (have 'n' spares)

Concrete cases

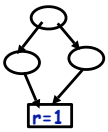
- What happens during crash happens during?
  Creating, moving, deleting, growing a file?
- How to deal with errors?
  The simplest approach: synchronous writes + fsck

Synchronous writes + fsck

- Synchronous writes = ordering state updates
  - **to do N modifications:**



  - **simple but slowwww.**
- fsck:
  - **After crash, sweep down entire FS tree, finding what is broken and try to fix.**
  - **Cost = O(size of FS). Yuck.**

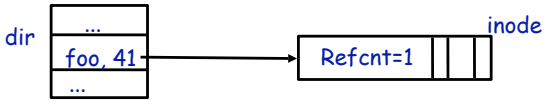


Unix file system invariants

- File and directory names are unique.
- All free objects are on free list.
  - **+ free list only holds free objects**
- Data blocks have exactly one pointer to them.
- Inode's ref count = the number of pointers to it.
- All objects are initialized.
  - **A new file should have no data blocks, a just allocated block should contain all zeros.**
- A crash can violate every one of these!

File creation

- open("foo", O_CREAT|O_RDWR|O_EXCL)
  - **1: search current working directory for "foo"**
    - **if found, return error (-EEXIST)**
    - **else find an empty slot**
  - **2: Find a free inode & mark as allocated.**
  - **3: Insert ("foo", inode #) into empty dir slot.**
  - **4: Write inode out to disk**



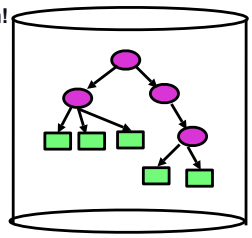
- Possible errors from crash?

Unused resources marked as "allocated"

- If free list assumed to be Truth, then many write order problems created.
  - **Rule:never persistently record a pointer to any object still on the free list.**
- Dual of allocation is deallocation. The problem happens there as well.
- Truncate:
  - **1: set pointer to block to 0.**
  - **2: put block on free list**
  - **if the writes for 1 & 2 get reversed, can falsely think something is freed**
  - **Dual rule: never reuse a resource before persistently nullifying all pointers to it.**

Reactive: reconstruct freelist on crash

- How?
  - **Mark and sweep garbage collection!**
  - **Start at root directory**
  - **Recursively traverse all objects, removing from free list.**



  - **Good: is a fixable error. Also fixes case of allocated objects marked as free.**
  - **Bad: Expensive. requires traversing all live objects and makes reboot slowwwww.**

Pointers to uninitialized data

- Crash happens between the time pointer to object recorded and object initialized.
- Uninitialized data?
  - **Security hole: Can see what was in there before.**
  - **Most file systems allow this, since expensive to prevent.**
- Much worse: Uninitialized meta data
  - **Filled with garbage. On a 4GB disk, what will 32-bit garbage block pointers look like?**
  - **Result: get control of disk blocks not supposed to have**
  - ***Major* security hole.**
  - **inode used to be a real inode? can see old file contents**
  - **inode points to blocks? Can view/modify other files**

Cannot fix, must prevent

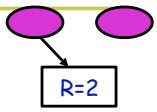
- Our rule:
  - **Never (persistently) point to a resource before it has been initialized.**
- Implication: file create 2 or 3 synchronous writes!
  - **Write 1: Write out freemap to disk. Wait.**
  - **Write 2: Write out 0s to initialize inode. Wait.**
  - **Write 3: Write out directory block containing pointer to inode. (maybe) Wait. (Why?)**

Deleting a file

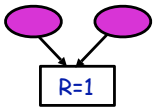
- Unlink("foo")
  - **1: traverse current working directory looking for "foo"**
    - **if not there, or wrong permissions, return error**
  - **2: clear directory entry**
  - **3: decrement inode's reference count**
  - **4: if count is zero, free inode and all blocks it points to**
- what happens if crash between 2&3, 3&4, after 4?

Bogus reference count

- Reference count too high?
  - **inode and its blocks will not be reclaimed**
    - **(2 gig file = big trouble)**
  - **what if we decrement count before removing pointer?**



- Reference count too low
  - **real danger: blocks will be marked free when still in use**
  - **major security hole: password file stored in "freed" blocks.**



- **Reactive: fix with mark & sweep**
- **Proactive: Never decrement reference counter before nullifying pointer to object.**

Proactive vs. reactive

- Proactive:
  - **Pays cost at each mutation, but crash recovery less expensive.**
  - **E.g., every time a block allocated or freed, have to synchronously write free list out.**
- Reactive: assumes crashes rare:
  - **Fix reference counts and reconstruct free list during recovery**
  - **Eliminates 1-2 disk writes per operation**

Growing a file

- write(fd, &c, 1)
  - **Translate current file position (byte offset) into location in inode (or indirect block, double indirect, …)**
  - **If meta data already points to a block, modify the block and write back.**
  - **Otherwise: (1) allocate a free block, (2) write out free list, (3) write out block, (4) write out pointer to block**
- What's bad things a crash can do?
- What about if we add block caching?
  - **"write back" cache? Orders can be flipped!**
  - **What's a bad thing to reverse?**

Moving a file

- mv foo bar (assume foo -> inode # 41)
  - **lookup "foo" in current working directory**
    - **if does not exist or wrong permissions, return error**
  - **lookup "bar" in current working directory**
    - **if wrong permissions, return error**
  - **1: nuke ("foo", inode 41)**
  - **2: insert ("bar", inode 41)**
  - **crash between 1 & 2?**
  - **what about if 2 and 1 get reordered?**

Conservatively moving a file

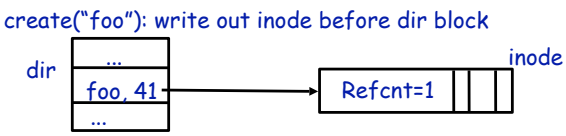
- Rule:
  - **never reset old pointer to object before a new pointer has been set**
- mv foo bar (assume foo -> inode # 41)
  - **lookup foo in current working directory**
    - **if does not exist or wrong permissions, return error**
  - **lookup bar in current working directory**
    - **if wrong permissions return error**
  - **0: increment inode 41's reference count. Wait.**
  - **1: insert ("bar", inode 41). Wait.**
  - **2: nuke ("foo", inode 41). Wait.**
  - **3: decrement inode 41's reference count**
- costly: 3 synchronous writes! How to exploit fsck?

Summary: the two fixable cases

- Case 1: Free list holds pointer to allocated block
  - **cause: crash during allocation or deallocation**
  - **rule: make free list conservative**
  - **free: nullify pointer before putting on free list**
  - **allocate: take off free list before adding pointer**
- Case 2: Wrong reference count
  - **too high = lost memory (but safe)**
  - **too low = reuse object still in use (very unsafe)**
  - **cause: crash while forming or removing a link**
  - **rule: conservatively set reference count to be high**
  - **unlink: nullify pointer before reference count decrement**
  - **link: increment reference count before adding pointer**
- Alternative: ignore rules and fix on reboot.

Summary: the two unfixable cases

- Case 1: Pointer to uninitialized (meta)data
  - **rule: initialize before writing out pointer to object**



  - **growing file? Typical: Hope crashes are rare…**
- Case 2: lost objects
  - **rule: never reset pointer before new pointer set**
  - **mv foo bar: create link "bar" before deleting link "foo."**
  - **crash during = too low refcnt, fix on reboot.**

4.4 BSD: fast file system (FFS)

- Reconstructs free list and reference counts on reboot
- Enforces two invariants:
  - **Directory names always reference valid inodes.**
  - **No block claimed by more than one inode.**
- Does this with three ordering rules:
  - **Write newly allocated inode to disk before name entered in directory.**
  - **Remove directory name before inode deallocated.**
  - **Write deallocated inode to disk before its blocks are placed on free list.**
- File creation and deletion take 2 synchronous writes
- Why does FFS need third rule? Inode recovery.

FFS: inode recovery

- Files can be lost if directory destroyed or crash happens before link can be set
  - **New twist: FFS can find lost inodes.**
- Facts:
  - **FFS pre-allocates inodes in known locations on disk.**
  - **Free inodes are to all 0s.**
- So?
  - **Fact 1 lets FFS find all inodes (whether or not there are any pointers to them)**
  - **Fact 2 tells FFS that any inode with non-zero contents is (probably) still in use.**
  - **fsck places unreferenced inodes with non-zero contents in the lost+found directory**

Fsck: reconstructing file system

```
# mark and sweep + fix reference counts
worklist := { root directory }
while e := pop(worklist)   # sweep down from roots
   foreach pointer p in e
        # if we haven't seen p and p contains pointers, add
        if p.type != Block and !seen{p}
                push(worklist, p);
                refs{p} = p.refcnt;    # p's notion of pointers to it
        seen{p} += 1;                    # count references to p
        freelist[p] = ALLOCATED;  # mark not free
foreach e in refs                        # fix reference counts
    if(seen{e} != refs{e})
        assert(p.type.has_refcnt);  # shouldn't happen
        e.refcnt = seen{e};
        e.dirty = true;
```

Write ordering

- Synchronous writes expensive
  - **sol'n have buffer cache provide ordering support**
- Whenever block "a" must be written before block "b" insert a dependency

  - **Before writing any block, check for dependencies (when deadlock?)**
- To eliminate dependency, synchronously write out each block in chain until done.
  - **Block B & C can be written immediately**
  - **Block A requires block B be synchronously written first.**

Write ordering problems

- Consider: create file, delete file. Circularity!