

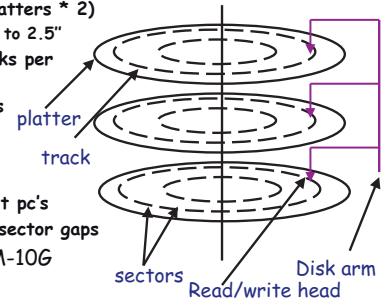
CS 140 Lecture 17: disk, files

Dawson Engler
Stanford CS department

The present

- ◆ A bottom up exposition of file systems
- ◆ Disk:
 - what it looks like
 - some implications
- ◆ Files:
 - what they look like
 - why
 - some implications
- ◆ The lecture ignores many other, richer storage system organizations (e.g., databases)

What do disks look like?

- ◆ 2-30 heads (platters * 2)
diameter 14" to 2.5"
 - ◆ 700-2048 tracks per surface
 - ◆ 16-160 sectors per track
 - ◆ sector size:
 - 64-8k bytes
 - 512 for most pc's
 - note: inter-sector gaps
 - ◆ capacity: 20M-10G
 - ◆ main adjectives: BIG, sloowwwwww
- 
- The diagram shows a cross-section of a disk platter with concentric tracks. A disk arm extends from the center, holding a read/write head that moves across the tracks. Labels include 'platter', 'track', 'sectors', 'Read/write head', and 'Disk arm'.

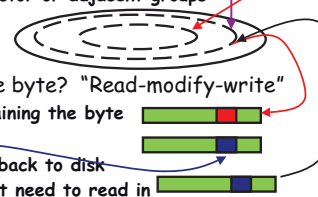
Example: a typical disk

- ◆ A typical sun disk (two, three years ago)
 - 1965 cylinders, 17 heads, 80 sec/track, 512 byte sectors
 - $(1965 * 17 * 80 * 512) / (1024 * 1024) = 1304$ MB
 - or about 700,000 double-spaced typewritten pages
 - platter RPM = 3600 RPM (some run at 7200RPM)
- ◆ Technology advances mostly in miniaturization
 - in 1975 40Mbytes took up space the size of a washing machine
 - today, my laptop has a 4GB disk.

Disk vs. Memory

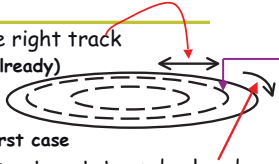
- | Disk | vs. | Memory |
|----------------------------------------------|-----|---------------------------------|
| ◆ Smallest write: sector | | ◆ (usually) bytes |
| ◆ Atomic write = sector | | ◆ byte, word |
| ◆ Random access: 10ms
not on a good curve | | ◆ 100 ns
faster all the time |
| ◆ Sequential access: 20MB/s | | ◆ 200-1000MB/s |
| ◆ Cost \$.02MB ('99 PC mag) | | ◆ \$1MB |
| ◆ Crash: doesn't matter ("non-volatile") | | ◆ contents gone ("volatile") |

Some useful facts

- ◆ Disk reads/writes in terms of sectors, not bytes
read/write single sector or adjacent groups
 - ◆ How to write a single byte? "Read-modify-write"
 - read in sector containing the byte
 - modify that byte
 - write entire sector back to disk
 - key: if cached, don't need to read in
 - ◆ Sector = unit of atomicity.
 - sector write done completely, even if crash in middle (disk saves up enough momentum to complete)
 - larger atomic units have to be synthesized by OS
- 
- The diagram shows a disk sector divided into several blocks. A red arrow points to one block, indicating a read operation. A blue arrow points to the same block, indicating a write operation. A red arrow points to the entire sector, indicating that the whole sector is written back to the disk. A blue arrow points to a different block, indicating a read operation for a byte that was not in the sector being written.

Some useful costs

- Seek: move disk arm to the right track
 - best case: 0ms (on track already)
 - worst: ~30-50ms (move over entire disk)
 - average: 10-20ms, 1/3 worst case
- Rotational delay: wait for sec to rotate under head
 - best: 0ms (over sector)
 - worst: ~16ms (entire rotation)
 - average: ~8ms (1/2 worst case)
- Transfer bandwidth: suck bits off of device
- Cost of disk access? Seek + rotation + transfer time
 - read a single sector: 10ms + 8ms + 50us ≈ 18ms
 - Cool: read an entire track? Seek + transfer! (why?)

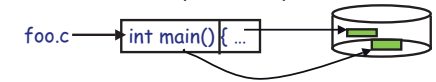


Some useful trends

- Disk bandwidth and cost/bit improving exponentially similar to CPU speed, memory size, etc.
- Seek time and rotational delay improving *very* slowly why? require moving physical object (disk arm)
- Some implications:
 - disk accesses a huge system bottleneck & getting worse
 - bandwidth increase lets system (pre-)fetch large chunks for about the same cost as small chunk.
 - Result? trade bandwidth for latency if you can get lots of related stuff.
 - How to get related stuff? Cluster together on disk
 - Memory size increasing faster than typical workload size
 - More and more of workload fits in file cache
 - disk traffic changes: mostly writes and new data

Files: named bytes on disk

- File abstraction:
 - user's view: named sequence of bytes
 - foo.c → int main() { ... }
 - FS's view: collection of disk blocks
 - file system's job: translate name & offset to disk blocks
 - offset:int → disk addr:int
- File operations:
 - create a file, delete a file
 - read from file, write to file
- Want: operations to have as few disk accesses as possible & have minimal space overhead



What's so hard about grouping blocks???

- In some sense, the problems we will look at are no different than those in virtual memory
 - like page tables, file system meta data are simply data structures used to construct mappings.
 - Page table: map virtual page # to physical page #
 - 28 → Page table → 33
 - file meta data: map byte offset to disk block address
 - 418 → Unix inode → 8003121
 - directory: map name to disk block address
 - foo.c → directory → 3330103

FS vs VM

- In some ways problem similar:
 - want location transparency, oblivious to size, & protection
- In some ways the problem is easier:
 - CPU time to do FS mappings not a big deal (= no TLB)
 - Page tables deal with sparse address spaces and random access, files are dense (0 .. filesize-1) & ~sequential
- In some way's problem is harder:
 - Each layer of translation = potential disk access
 - Space a huge premium! (But disk is huge?!?) Reason? Cache space never enough, the amount of data you can get into one fetch never enough.
 - Range very extreme: Many <10k, some more than GB.
 - Implications?

Some working intuitions

- FS performance dominated by # of disk accesses
 - Each access costs 10s of milliseconds
 - Touch the disk 50-100 extra times = 1 *second*
 - Can easily do 100s of millions of ALU ops in same time
- Access cost dominated by movement, not transfer
 - Can get 20x the data for only ~5% more overhead
 - 1 sector = 10ms + 8ms + 50us (512/10MB/s) = 18ms
 - 20 sectors = 10ms + 8ms + 1ms = 19ms
- Observations:
 - all blocks in file tend to be used together, sequentially
 - all files in a directory tend to be used together
 - all names in a directory tend to be used together
 - How to exploit?

Common addressing patterns

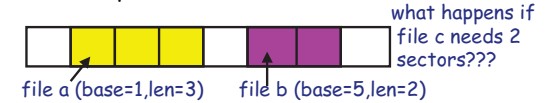
- ◆ Sequential:
 - file data processed in sequential order
 - by far the most common mode
 - example: editor writes out new file, compiler reads in file, etc.
- ◆ Random access:
 - address any block in file directly without passing through predecessors
 - examples: data set for demand paging, databases
- ◆ Keyed access:
 - search for block with particular values
 - examples: associative data base, index
 - usually not provided by OS

Problem: how to track file's data?

- ◆ Disk management:
 - Need to keep track of where file contents are on disk
 - Must be able to use this to map byte offset to disk block
 - The data structure used to track a file's sectors is called a **file descriptor**
 - file descriptors often stored on disk along with file
 - ◆ Things to keep in mind while designing file structure:
 - Most files are small
 - Much of the disk is allocated to large files
 - Many of the I/O operations are made to large files
- Want good sequential and good random access (what do these require?)

Simple mechanism: contiguous allocation

- ◆ "Extent-based": allocate files like segmented memory
 - When creating a file, make the user specify pre-specify its length and allocate all space at once
 - File descriptor contents: location and size



Example: IBM OS/360

- Pro: simple, fast access, both sequential and random.
- Cons? (What does VM scheme does this correspond to?)

Linked files

- ◆ Basically a linked list on disk.
 - Keep a linked list of all free blocks
 - file descriptor contents: a pointer to file's first block
 - in each block, keep a pointer to the next one
-
- how do you find the last block in a?
- file a (base=1) file b (base=5)
- pro: easy dynamic growth & sequential access, no fragmentation
con?
Examples (sort-of): Alto, TOPS-10, DOS FAT

Example: DOS FS (simplified)

- ◆ Uses linked files. Cute: links reside in fixed-sized "file allocation table" (FAT) rather than in the blocks.
- FAT (16-bit entries)
- | | | |
|---------------|-----|------|
| Directory (5) | 0 | free |
| | 1 | eof |
| a: 6 | 2 | 1 |
| b: 2 | 3 | eof |
| | 4 | 3 |
| | 5 | eof |
| | 6 | 4 |
| | ... | ... |
- file a: 6 → 4 → 3
- file b: 2 → 1
- Still do pointer chasing, but can cache entire FAT so can be cheap compared to disk access.

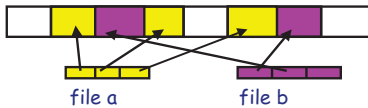
FAT discussion

- ◆ Entry size = 16 bits
 - What's the maximum size of the FAT?
 - Given a 512 byte block, what's the maximum size of FS?
 - One attack: go to bigger blocks. Pro? Con?
- ◆ Space overhead of FAT is trivial:
 - 2 bytes / 512 byte block = ~.4% (Compare to Unix)
- ◆ Reliability: how to protect against errors?
 - Create duplicate copies of FAT on disk.
 - State duplication a very common theme in reliability
- ◆ Bootstrapping: where is root directory?
 - Fixed location on disk:

FAT	(opt) FAT	root dir	...
-----	-----------	----------	-----

Indexed files

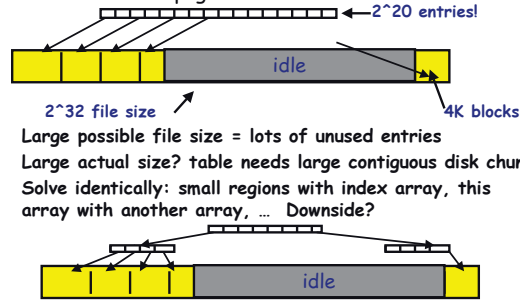
- Each file has an array holding all of its block pointers (purpose and issues = those of a page table)
- max file size fixed by array's size (static or dynamic?)
- create: allocate array to hold all file's blocks, but allocate on demand using free list



pro: both sequential and random access easy
con?

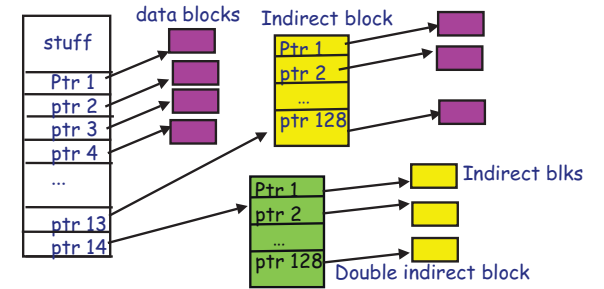
Indexed files

- Issues same as in page tables



Multi-level indexed files: ~4.3 BSD

- File descriptor (inode) = 14 block pointers + "stuff"



Unix discussion

- Pro?
 - simple, easy to build, fast access to small files
 - Maximum file length fixed, but large. (With 4k blks?)
- Cons:
 - what's the worst case # of accesses?
 - What's some bad space overheads?
- An empirical problem:
 - because you allocate blocks by taking them off unordered freelist, meta data and data get strewn across disk

More about inodes

- Inodes are stored in a fixed sized array
 - Size of array determined when disk is initialized and can't be changed. Array lives in known location on disk. Originally at one side of disk:
-
- Now is smeared across it (why?)
-
- The index of an inode in the inode array called an i-number. Internally, the OS refers to files by inumber
 - When file is opened, the inode brought in memory, when closed, it is flushed back to disk.

Example: Unix file system

- Want to modify byte 4 in /a/b.c:
 - readin root **directory** (blk 10)
 - lookup a (blk 12); readin
 - lookup **inode** for b.c (13); readin
-
- use inode to find blk for byte 4 (blksize = 512, so offset = 0 gives blk 14); readin and modify

Turn off comments!

CS 140 Lecture 19: (FS) consistency

Dawson Engler
Stanford CS department

Last time, this time

- ◆ The Big News: Systems crash
 - problem 1: blows away "volatile state" (main memory)
 - problem 2: interrupts updates to "stable state" (disk)
- ◆ Last time:
 - Used state duplication and idempotent actions to create arbitrary sized atomic disk updates.
 - Today: recovery for atomic write
- ◆ This time: Atomicity and consistency in file systems
- ◆ Basic idea:
 - limit error states by updating disk in stylized way
 - after crash run recovery program that knows how to take each error state to a valid state
 - careful: can crash during recovery!

SABRE atomic disk operations

```
void atomic-put(data)          blk atomic-get()
  version++; # unique integer  V1data := get(V1)
  put(version, V1);            D1data := get(D1);
  put(data, D1);              V2data := get(V2);
  put(version, V2);           D2data := get(D2);
  put(data, D2);              if(V1 data == V2)
                              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" }

If we don't fix after crash, what can happen to make us start using a wrong value? Break into two cases: (1) we are taking last value, does it matter if we corrupt? (Only recycle version) (2) if we are using first? (yupper)

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")
Could crash and get in good state again 2-3-4
 - ◆ There are 6 cases, depending on where we failed in atomic-put:
Actually, this isn't true: what other cases are there? (failure between!)
- | 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"} | <small>Error if recycle version</small> |
| the third | {#3, "seat 31", #2.5, "seat 25"} | |
| the fourth | {#3, "seat 31", #3, "seat 35"} | <small>Error if crash on version mod</small> |
| after | {#3, "seat 31", #3, "seat 31"} | |

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); Is it ok to corrupt d2?
    else
      # if we crash and corrupt D1 will get back here
      put(D2data, D1); Is it ok to corrupt d1?
      # if we crash and corrupt V1, will get back here
      put(V2data, V1); Is it ok to corrupt 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)

Fighting failure

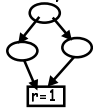
- ◆ In general, coping with failure consists of first defining a failure model composed of
 - **Acceptable failures.** E.g., the earth is destroyed by weirdos from Mars. The loss of a file viewed as unavoidable.
 - **Unacceptable failures.** E.g. power outage: lost file not ok
- ◆ Second, devise a recovery 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?

The rest today: concrete cases

- ◆ What happens during crash happens during
 - creating, moving, deleting, growing a file?
- ◆ Woven throughout: how to deal with errors
 - the simplest approach: synchronous writes + fsck

Synchronous writes + fsck

- ◆ Synchronous writes = ordering state updates
 - to do n modifications:
 - write block 1 → Wait for disk
 - write block 2 → Wait for disk
 - ...
 - simple but slowww.
- ◆ fsck:
 - after crash, sweep down entire FS tree, finding what is broken and try to fix.
- Cost = $O(\text{size of FS})$. Yuck.

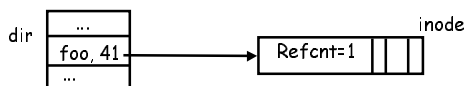


Unix file system invariants

- ◆ 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"

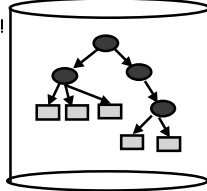
- ◆ If free list assumed to be Truth, then many write order problems created.
 - Rule: never persistently record a pointer to any object still on the free list
- ◆ Dual of allocation is deallocation. The problem happens there as well.
- ◆ Truncate:
 - 1: set pointer to block to 0.
 - 2: put block on 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.

Does a cycle matter? If we can have links to files, does this create cycles? What does?

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

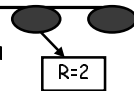
◆ what happens if crash between 2&3, 3&4, after 4?

Could have thought you'd deleted a file, but after a crash it comes back.

Bogus reference count

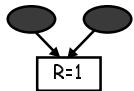
◆ Reference count too high?

- 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?

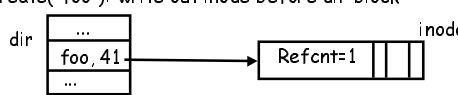
Conservatively moving a file

- ◆ 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
 - `create("foo")`: write out inode before dir block

The diagram shows a directory entry 'foo, 41' in a 'dir' box. An arrow points from this entry to an 'inode' box containing 'Refcnt=1' and three empty slots.

- 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

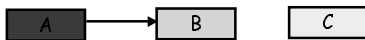
- ◆ Files can be lost if directory destroyed or crash happens before link can be set
 - New twist: FFS can find lost inodes
- ◆ Facts:
 - FFS pre-allocates inodes in known locations on disk
 - Free inodes are to all 0s.
- ◆ So?
 - Fact 1 lets FFS find all inodes (whether or not there are any pointers 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.

CS 140 Lecture 20: SPEED! SPEED! SPEED!

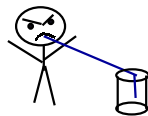
Dawson Engler
Stanford CS department

Past, Present & Future

- Recent past: dealing with failure.
- Recent present (today): dealing with huge disk latencies
- Recent future (monday): dealing with reality: actual file systems dealing with multiple disks (raid)
- Further future: next week: networking
- Readings (all on course web page): A Fast File System for UNIX (paper) Unix implementation paper

Sucking + blowing data through slow straws

- Latency tricks
 - Caching (data centric)
 - Code migration (code centric)
 - Prefetching (dual: asynchronous writes)
 - Clever data layout
- Throughput tricks:
 - Increase bandwidth? Duplicate device N times
 - Hide latency? Run another thread while waiting
 - "Money can buy bandwidth. Latency requires bribing God."
- These tricks are eternal themes
 - use whenever need fast access to data living in slow place
 - common straws: I/O bus, memory bus, network, space (bi-directional), time (uni-directional)



Caching: the buffer cache

- Our cliché: past predicts future? Use a cache.
 - Blk 514 → Blk 1111 → Blk 201
 - Blk 202 → Blk 1112
 - Blk 203 → Blk 1113
- Buffer cache roughly similar to page cache
 - good: memory growth on similar curve as disk capacity
 - bad: forces 2 copies for normal interface; "cache wiping"
- Our timeless cache questions:
 - How big? How to evict? What happens to writes? What to prefetch?

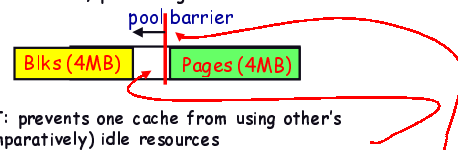
Decision #1: Size

- How big to make cache?
 - Main issue: partitioning memory buffer cache and VM page cache (its main competitor)
 - Early systems: fixed-size cache.
- Problem: doesn't adapt to workload.
 - Obvious sol'n: variable sized cache.
- Problem: enormous files not uncommon.
 - Solution: fixed size caches?
 - Same problem sharing page cache across "n" users. Solve in same way (with the same slide!)



Split versus shared caches

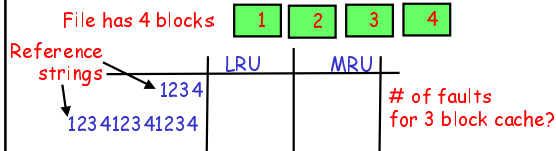
- Private caches
 - block cache and page cache have a separate pool of pages
 - miss in one can only replace one of its own pages
 - isolates cache, preventing interference
- BUT: prevents one cache from using other's (comparatively) idle resources
 - efficient memory usage needs way to (slowly) change the allocations to each pool. Usually keep a fixed reserve
 - Qs: What is "slowly"? How big a pool? When to migrate?



Decision #2: Eviction policy

- ◆ "Dance with the one that brought you"
 - The reason we used a cache was because past ~ future.
 - So use LRU (as usual). New twist: can have perfect LRU!
 - New twist #2: Unfortunately, now it's less of a good idea since many files much bigger than VM objects.

- ◆ What to do when file larger than cache?
 - LRU = exact worse thing. !LRU = best thing! (MRU)



Decision #3: Write back policies

- ◆ Importance of writes?
 

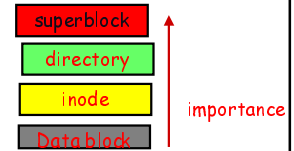
- ◆ Write through (simple, but slow):
 - whenever modify cached block, write block to disk.
 - Con: makes writes slow.
 - Pro: keeps FS in consistent state. PC OSES do this.
- ◆ Write back (faster, but complex (and dangerous)):
 - whenever modify block, mark as dirty. Flushed later
 - pro: fast writes, absorbs writes, enables batching
 - con: More you defer write back, the worse a crash is.

Write back complications

- ◆ Fundamental tension:
 - On crash, all modified data in cache is lost.
 - Longer you postpone write back, the worst the damage is, but the faster you are.
- ◆ Four times to flush:
 - when block evicted (this is ~ what VM does)
 - when a file is closed (distributed file systems do this)
 - on an explicit flush ("man sync")
 - when a time interval elapses (30 secs in Unix)
- ◆ Write back doesn't respect ordering
 - crash = file in weird jumbled state
- ◆ Finally: OS may not have any choice!
 - Disk could be doing write buffering.

Flushing nuance: All blocks are not equal

- ◆ Losing some blocks worse than losing others. Usually:



- ◆ File system effects:
 - flush modified meta data back quickly
 - Note: was an implicit side-effect of synchronous writes for meta data mods
- ◆ Application effects:
 - have a really important file? Issue a manual flush (sync) to make sure its saved to disk.
 - Databases can be forced to do this. Frequently, they will just by-pass file system.

Decision #4: what to prefetch?

- ◆ Optimal: the blocks we need are fetched in just enough time for us to use them.
 - Note: if we had enough disk bandwidth and ability to predict future wouldn't need (much of) a cache: just fetch block that will be used, and then throw way.
- ◆ Example:
 - App issues reads disk block every 1ms
 - Disk can accept new request every 1ms
 - Each request takes 10ms to satisfy
 - How big a cache do we need?
- ◆ The usual problem: we don't know the future.
 - (Cache = trade space for stupidity)

Location ~ future ("Spatial locality")

- ◆ "Spatial locality"
 - Access one object, will probably access ones close to it
 - E.g., read logical block in file, probably read next too.
 - So, when get one block, get the next n that follow.
 - More precise: when access one object, will probably access ones *related* to it. So, cluster them, and use above scheme.
- ◆ How big is "n"?
 - Tradeoff: larger n means more payoff if we are right, but more wastage if not. (Usually n < 65k bytes)
 - Variant: get big chunks & preferentially discard

Decision #5: scheduling disk arm

- ◆ One resource, multiple requests = scheduling problem
one disk arm, multiple read and write requests
- ◆ Goals?
 - minimize seeks
 - no starvation
- ◆ Disk scheduling algorithms have close analogues in process scheduling
and, similarly, become more important as more requests.
(However, unlike job queue, disk queue usually short...)
- ◆ Next: three scheduling variations
 - optimize seek times
 - newer disks require taking account of rotational delay

First-come-first-served

- ◆ Schedule disk requests in order received.
Fair but may have huge seeks for no good reason.
 - ◆ Example: read from cylinders 0, 100, 1, 101, 2, 102
-
- ◆ total seek distance?

Shortest seek time first

- ◆ Handle nearest request first
optimize disk arm movement, but can be (really) unfair
 - ◆ Example: read from cylinders 0, 100, 1, 101, 2, 102
-
- ◆ Total seek distance?
 - ◆ SSTF = shortest-time-to-completion!
Why can we do STTC? We know job length.

Scan (elevator algorithm)

- ◆ Move arm back & forth, handling requests underneath
more fair to requests scattered across disk, similar performance to SSTF
 - ◆ Example: read from cylinders 0, 100, 1, 101, 2, 102
-
- ◆ Total seek distance?
 - ◆ works well when many requests
if not many, about 1/2 time won't get shortest seek

Original Unix File System

- ◆ Simple and elegant:
-
- ◆ Nouns:
 - data blocks
 - inodes (directories represented as files)
 - hard links
 - superblock. (specifies number of blks in FS, counts of max # of files, pointer to head of free list)
 - ◆ Problem: slow
only gets 20Kb/sec (2% of disk maximum) even for sequential disk transfers!

A plethora of performance costs

- ◆ Blocks too small (512 bytes)
 - file index too large
 - too many layers of mapping indirection
 - transfer rate low (get one block at a time)
- ◆ Sucky clustering of related objects:
 - Consecutive file blocks not close together
 - Inodes far from data blocks
 - Inodes for directory not close together
 - poor enumeration performance: e.g., "ls", "grep foo *.c"
- ◆ Next: how FFS fixes these problems (to a degree)

Problem 1: Too small block size

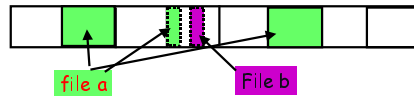
- Why not just make bigger?

Block size	space wasted	file bandwidth
512	6.9%	2.6%
1024	11.8%	3.3%
2048	22.4%	6.4%
4096	45.6%	12.0%
1MB	99.0%	97.2%

- Bigger block increases bandwidth, but how to deal with wastage ("internal fragmentation")?
Use idea from malloc: split unused portion.

Handling internal fragmentation

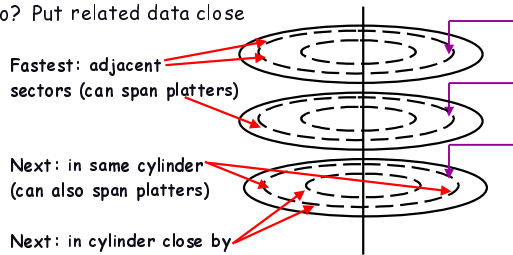
- BSD FFS:
 - has large block size (4096 or 8192)
 - allow large blocks to be chopped into small ones ("fragments")
 - Used for little files and pieces at the ends of files



- Best way to eliminate internal fragmentation?
Variable sized splits of course
Why does FFS use fixed-sized fragments (1024, 2048)?

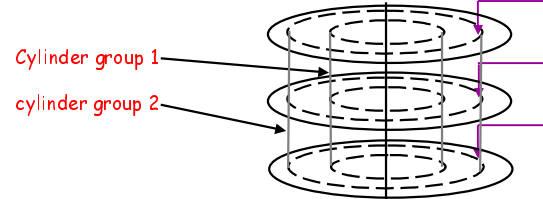
Prob' 2: Where to allocate data?

- Our central fact:
Moving disk head expensive
- So? Put related data close



Clustering related objects in FFS

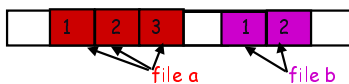
- 1 or more consecutive cylinders into a "cylinder group"



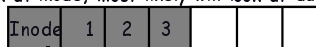
Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
Tries to put everything related in same cylinder group
Tries to put everything not related in different group (!)

Clustering in FFS

- Tries to put sequential blocks in adjacent sectors (access one block, probably access next)



- Tries to keep inode in same cylinder as file data: (if you look at inode, most likely will look at data too)



- Tries to keep all inodes in a dir in same cylinder group (access one name, frequently access many)
"ls -l"

What's a cylinder group look like?

- Basically a mini-Unix file system:



superblock

- How how to ensure there's space for related stuff?
Place different directories in different cylinder groups
Keep a "free space reserve" so can allocate near existing things
when file grows to big (1MB) send its remainder to different cylinder group.

Prob' 3: Finding space for related objects

- ◆ Old Unix (& dos): Linked list of free blocks
Just take a block off of the head. Easy.



Bad: free list gets jumbled over time. Finding adjacent blocks hard and slow

- ◆ FFS: switch to bit-map of free blocks
10101011111100000111111000101100
easier to find contiguous blocks.
Small, so usually keep entire thing in memory
key: keep a reserve of free blocks. Makes finding a close block easier

Using a bitmap

- ◆ Usually keep entire bitmap in memory:
4G disk / 4K byte blocks. How big is map?
- ◆ Allocate block close to block x?
check for blocks near $bmap[x/32]$
if disk almost empty, will likely find one near
as disk becomes full, search becomes more expensive and less effective.
- ◆ Trade space for time (search time, file access time)
keep a reserve (e.g., 10%) of disk always free, ideally scattered across disk
don't tell users (df --> 110% full)
N platters = N adjacent blocks
with 10% free, can almost always find one of them free

So what did we gain?

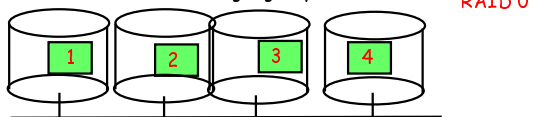
- ◆ Performance improvements:
able to get 20-40% of disk bandwidth for large files
10-20x original Unix file system!
Better small file performance (why?)
- ◆ Is this the best we can do? No.
- ◆ Block based rather than extent based
name contiguous blocks with single pointer and length
(Linux ext2fs)
- ◆ Writes of meta data done synchronously
really hurts small file performance
make asynchronous with write-ordering ("soft updates")
or logging (the episode file system, ~LFS)
play with semantics (/tmp file systems)

RAID: exploiting multiple disks

- ◆ "Redundant array of inexpensive disks"
(acronym from same people that named RISC)
- ◆ Empirical observation:
to get given capacity much cheaper to buy n small disks
than 1 large one.
Once you buy multiple disks, can do some neat tricks
- ◆ Trick 1: n-fold bandwidth increase (ideal)
stripe data across multiple disks
read/write from all simultaneously
(common theme: money buys bandwidth by buying multiple
straws (networks, disks, highway lanes,...))
- ◆ Trick 2: use to increase reliability
keep copies of data on different disks. Switch on failure

Theme 1 in RAID: Major bandwidth

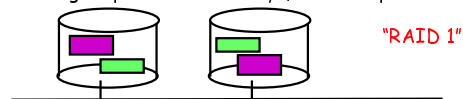
- ◆ Striping: smear data across all disks.
Ideal: N reads/writes going in parallel



- ◆ Basic idea: straw too thin? Buy more of them and suck in parallel.
- ◆ Problem: reliability.
N disks, probability that one blows up increases as well.
Independent failures? 100 disks = 100 more likely that one is down. MTTF of 200K hrs/100 = 2Khrs ~ 3months

Theme 2 in RAID: Reliability

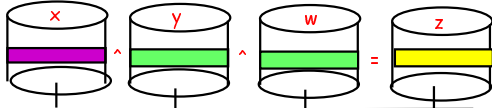
- ◆ Mirroring: improves reliability (and read performance)



- every read/write operation issued to both disks
- ◆ Basic idea (seen before):
duplicate state to increase reliability
- ◆ Like backups except:
copy is completely up to data
"crossover" is fast
fault-tolerant systems use this basic trick a lot

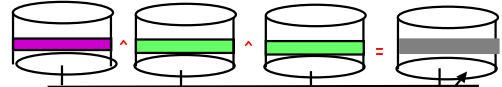
Cute xor tricks

- ◆ Recall:
 - $x \oplus x = 0$
 - if $z = x \oplus y$ then: $x = z \oplus y$ and $y = z \oplus x$
 - proof: $x = x \oplus y \oplus y$, but $x \oplus y = z$ so $(x \oplus y) \oplus y = z \oplus y$
- ◆ So? Instead of mirroring, use one disk to hold z!
 $z = \text{xor of all other disks}$



recover from any single failure by xoring the remaining disks with this "parity" disk.

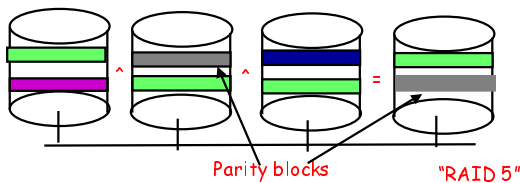
Block-interleaved parity tradeoffs



- + less space than mirroring
- + much better bandwidth in ideal case

- ◆ Prob 1: "the small write problem" "RAID 4"
 - Must recalculate parity on each write.
 - If write entire stripe, no problem. If not writing 1 block requires 2 reads + 2 writes!
- ◆ Prob 2: parity disk = bottleneck
 - If one disk used for parity, it will always be busy
 - sol'n: let the parity block in each group float around.

Block-interleaved Distributed parity



- + eliminates parity disk bottleneck
- + all disks can participate in read requests
- best small read, large read, & large write performance (still whipped by mirroring on writes)

Reliability variant: write-ahead logging

- ◆ We ordered file system operations to guard against corruption.
- ◆ Instead:
 - Create a record(s) of what we intend to do and append to end of a **log** (sequential array of records).
 - Then write updated log to disk. Then do the action
 - After crash, start at beginning of log, doing each action. Actions must be **idempotent**. Allows us to keep **replaying** log until we consume it, no matter how many times we're interrupted by a crash.
 - Optimization: allow recovery to skip records by grouping records into **transactions** and placing a **commit** entry in log when all actions in a transaction have been successfully performed.

Example: write-ahead logging

- ◆ to create a file foo.c write two log records
- | |
|------------------------------------------------------|
| ... |
| ... |
| write ("foo", 51) @ byte 30 in blk 319 # create link |
| write 0s into blk 51 # initialize inode |

Log grows downward

- ◆ Write log to disk. Then modify cached copies of blks 319 and 51. After these written to disk, write commit.
- ◆ Key: brute writes are always idempotent.
 E.g., can repeat "i = 5" as often as you want.

What have we gained?

- ◆ Result 1: freedom from ordering tyranny
 - If log holds records for many dependencies, a single disk write eliminates many sequential dependencies.
 - Can then write blocks containing actual directories and inodes in any order (key: don't have to wait 10ms before writing the next)
- ◆ Result 2: fast recovery
 - Rather than scan entire disk checking for errors, just replay log. Much much faster the episode file system does this.
- ◆ Result 3: compressed dependency representation
 - if the machine has small amount of non-volatile RAM, keep log in it. Potentially eliminate ***all*** ordered writes!