6.824 2006 Lecture 6: Crash Recovery for Disk File Systems

Common theme in system-building:
  You have some persistent state you're maintaining, reading and writing.
    Maybe the data is replicated in multiple servers' memory,
    maybe it's on one or more disks.
  You use caching and fancy data structures for efficiency.
  The hard part always turns out to be recovery after a crash.

Goals:
  1. maintain storage system's internal invariants
  2. preserve ordered prefix of user's operations

Most solutions have a similar flavor:
  1. each operation takes storage from legal state to legal state,
     perhaps w/ multiple updates to storage system
  2. order updates to persistent store so there are commit points
  3. recovery procedure can finish or un-do partial operations.
  it all has to be fast! persistent storage has always been slow.

Case study: disk file systems.
  Critical for performance and crash recovery of individual machines.
  Interacts with distributed protocols, for both reasons.
  Crash recovery techniques similar to those in distributed systems.
  Trade-offs are often the same (performance vs durability).

A file system is a fairly complex abstract data structure:
  (this is for UNIX)
  tree rooted at root i-node
  directory entries
  file/subdirectory i-nodes
  file blocks
  file block number lists

i-nodes and directory contents usually called meta-data.
  As opposed to file contents.

Even at this abstract level there are crash recovery problems:
  What if you crash in the middle of a rename()?

But there is more:
  These objects live somewhere on the disk.
  [circle with i-node, data, &c]
  The file system objects have disk block addresses. Sector number?
  File system must allocate disk blocks for new i-nodes &c.
    Someone decides where to place i-nodes.
  File system must release unused blocks after deletion.
  So there must be a list of free disk blocks.
  And you don't want allocated block on free list!
  Will it be expensive to keep free list up to date on the disk?

What does recovery do?
  For example, UNIX fsck program that runs at boot-time.
  Very similar to a mark-and-sweep garbage collector.
  Descends tree, remembers all allocated i-nodes and blocks.
  All others must be free, so fsck just re-initializes free lists.

Cite as: Robert Morris, course materials for 6.824 Distributed Computer Systems Engineering, Spring 2006. MIT OpenCourseWare (http://ocw.mit.edu/), Massachusetts Institute of Technology. Downloaded on [DD Month YYYY].

Also checks for block used by two files, file length != number of blocks, &c.
Maybe find problem it can't fix (which file should get the doubly-used block).
Asks the user.
Goal: finish or un-do ops in progress at the time of the crash.
Leave file system in a state that it could have been in if the crash had
happened ether before or after last user op.
So perhaps user loses last few ops, no other problems.
Or notify the user if that's not possible.

Final ingredient:
Kernel's in-memory disk buffer cache of recently used blocks.
Hugely effective for reads (all those root i-node accesses).
The result is that the bottleneck is often disk writes.
So disk caches are also usually write-*back*: they hold dirty blocks.
Dirty blocks are lost if there's a crash!

What are the specific problems?

Example:
fd = create("d/f", 0666);
write(fd, "hello", 5);
Ignore reads since cached, here are the block writes:
1. i-node free bit-map (get a free i-node for f)
2. f's i-node (write owner &c)
3. d's contents (add "f" -> i-number mapping)
4. d's i-node (longer length, mtime)
5. block free bit-map (get a free i-node for f's data)
6. data block
7. f's i-node (add block to list, update mtime and length)

How fast can we create small files?
If each write goes to disk, then 70 ms/file, or 14 files/second.
Pretty slow if you are are un-tarring a big tar file.
If FS only writes into disk cache, very quickly.
But cache will eventually fill up with dirty blocks, must write to disk.
Then writes 1, 2, 3, 4, 5, and 7 are amortized over many files
But write 6 is one per file.
Sustained rate of maybe 100 files/second. 10x faster than sync writes.
So you would like write-back!

And unlink:
unlink("d/f");
8. d's contents
9. d's i-node (mtime)
10. free i-node bitmap
11. free block bitmap

Can we recover sensibly with a write-back cache?
The cache module may write to disk in any order.
The game: a few dirty blocks flushed to disk, then crash, recovery.
Example: 1-7 and 8, recovery sees unused i-node, frees it.

Cite as: Robert Morris, course materials for 6.824 Distributed Computer Systems Engineering, Spring 2006. MIT OpenCourseWare (http://ocw.mit.edu/), Massachusetts Institute of Technology. Downloaded on [DD Month YYYY].

Example: just 3, recovery sees used i-node on free list; ask the user?
Example: 1-7 and 10, recovery sees used i-node on free list; ask the user?
These are benign, though annoying for the user.
Clearly there's a vast number of outcomes.
Are they all benign?

Here's the worst:
unlink("f1");
create("f2");
Create happens to re-use the i-node freed by the unlink.
Suppose only create write #3 goes to disk, but none of the unlink's writes
Crash.
After re-start, what does recovery see?
The file system looks correct! Nothing to fix.
But file f1 actually has file f2's contents!
Serious *undetected* inconsisency.
This is *not* a state the file system could have been in if the crash had occured slightly earlier or later.
We didn't just lose the last few updates.
And fsck did not notify the user there was an unfixable problem!

How can we avoid this delete/create inconsistency?
Observation: we only care about what's visible in the file system tree.
Goal: on-disk directory entry must always point to correct on-disk i-node.
Unlink rule: remove dirent *on disk* before freeing i-node.
Create rule: initialize new i-node *on disk* before creating directory entry.
In general, directory entry writes should be commit points.
Crash just before leaves us with unused allocated i-node.
Crash just after is fine.

Synchronous disk writes in the order I gave is sufficient.
For most file system operations, there is some recoverable synchronous order.
Because the file system is a tree, you can prepare the new sub-tree and cause it to appear in the old one with one operation.
And most operations are "small", just affect leaves.
What about rename()?

Can we eliminate some of the sync writes in file creation?
To speed up file creation.
What ordering constraints can we identify?
#2 before #3
#6 before #7
#3 before #4? hard to say, maybe fsck can correct length, but not mtime.
#1 and #5 need never occur, since fsck recovers it.
#3 can be deferred, since it is a commit point once #2 has completed.
So perhaps only #2 and #6 need to be synchronous.
To force them to occur before #3 and #7.
Perhaps #3 to force it to happen before #4.
UNIX: #2, #3, but not #6.

Cite as: Robert Morris, course materials for 6.824 Distributed Computer Systems Engineering, Spring 2006. MIT OpenCourseWare (http://ocw.mit.edu/), Massachusetts Institute of Technology. Downloaded on [DD Month YYYY].

```
    It defends only its own meta-data, not your file data.
    Use fsync() if you care.

Performance: two disk writes/seeks for a file create, so 50
files/second.

Can we do better?
```

Cite as: Robert Morris, course materials for 6.824 Distributed Computer Systems Engineering, Spring 2006. MIT OpenCourseWare (http://ocw.mit.edu/), Massachusetts Institute of Technology. Downloaded on [DD Month YYYY].