Stanford University
Computer Science Department
CS 240 Quiz 2
Fall 2005

November 28, 2005

This is an open-book exam. You have 50 minutes to answer 8 out of 10 questions. Write all of your answers directly on the paper. Make your answers as concise as possible. Sentence fragments ok.

NOTE: We will take off points if a correct answer also includes incorrect or irrelevant information. (I.e., don't put in everything you know in hopes of saying the correct buzzword.)

Question	Score
1 - 5	
6 - 10	
total	

Stanford University Honor Code

In accordance with both the letter and the spirit of the Honor Code, I did not cheat on this exam nor will I assist someone else cheating.

Name and Stanford ID:

Signature:

Answer eight of the following ten questions and, in a sentence or two, say *why* your answer holds (5 points each).

1. Assume you have an infinitely fast CPU. Describe what the MLFRR will be.
   Infinitely fast CPU = processing packets is free = you should not be able to induce starvation. Your MLFRR should be the speed at which you can receive/transmit packets. If you receive more quickly than you can send, then there will be loses at the xmit queue. If you receive at the same or lower rate than you xmit you won't have loses internally, but won't be able to exceed min(receive, xmit) rate in any case.

2. You are hired to be a livelock doctor. Give three symptoms you could reasonably expect to see in a livelocked system (also: give intuition).
   Livelock is going to cause things to starve and packets to be dropped internally (the starved thing won't run, so can't eat packets, and queues are finite size). Starvation: applications not running, xmit or receive paths not running. Packets being dropped off of queues. Output not the right level.

3. Give an end-to-end style argument to justify why map/reduce's solution to problematic input is reasonable.
   It will skip inputs that cause its code to blow up. You could argue that you do not need a non-zero error rate because the probability that a machine is down when you try to crawl it is not zero: end to end would say that these only have to be roughly equivalent. An alternative argument is: Google applications are mostly statistical applications and they can tolerate errors.

4. NFS: Your ex-cs140 partner has stalking problems again and confronts you with a tricky bug: His KewlDewdSH shell started giving him "Permission Denied" errors while writing to a temp file when before he never had a problem. You recall that he recently switched to running his shell over NFS, what likely is causing the problem?
   NSF does not know what files are "open." Therefore it has different semantics than local Unix file system. There are three possible answers: 1) If you open a file, then change its permission to be not writable, you will still be able to write to it locally, but on NFS you'll get a permission denied error. 2) if you delete an open file you will still be able to access it locally, but on NFS the thing will be blown away. You probably won't get a permission denied error, but a not-exist one though. 3) NFS does root squashing by default. Root uid(0) in client requests are replaced with nobody uid(-1). This can also cause a permission denied error. (actually, there is also an option for NFS to squash all user ids to -1)

5. You want to model-check NFS, LBFS, and LFS. Define the StableFS for each assuming there is only a single, single-threaded client running. (Note: be as precise as you can, but a exact description might not always be possible.)
   StableFS = what the file system should recover to after crash. NSF does each operation synchronously, so should be the last operation that was received and acted on.

(VolatileFS = StableFS plus or minus the last operation) LBFS follows NSF except that it uses "real" close-open consistency. So, the StableFS should be the same as the VolatileFS plus or minus the last file closed. Without rollforward, LFS's StableFS is just whatever is in the last checkpoint. With it, its whatever you get by rolling forward (correctly).

6. Assume that LFS inodes use both indirect and double indirect blocks to track the blocks in a file. What is the worst case number of segments that you might have to read to access a single block in a file? What is a simple set of operations to cause this?
Reading last block in a large file = read inode, double indirect, single indirect block. These can be scattered around 4 segments. Easy way: write block (all four in segment). Wait until segment written. Now write adjacent block. (This will pull inode, double, single into another segment.) Wait until segment written. Now write a block that needs same double but different single, (this pulls double into another.) Wait until segment written. Now write some other direct block. Wait. (Pulls inode into another). This assumes you already have the inode number of the file and the inode map block is already cached in memory. Otherwise, you may read more than 4 segments.

7. Leases: accurately redraw Figure 2 assuming at least one leaseholder was down. (Note, make sure your drawing takes into account what happens when (1) S=1 and (2) as the lease term grows.)
*If we are not sharing (S=1), then should just be us. Otherwise the line should be equal to read delay * read request percentage + (lease term + network propagation delay) * p * write request percentage. Read delay can be read from the original graph in the paper. The paper also mentioned that 1/20 of the requests are writes. p ($0 \leq p \leq 1$) should be some expected value derived from poison distribution and S: when a write request comes in, the dead lease holder may already own the lease for a while, so the writer doesn't need to wait for an entire lease term. Deriving p is outside the scope of the class, so we give full points to the answers that didn't consider p.*

8. LBFS: program P on client A opens file F and starts using it. The server then revokes F's lease from A. P then does a read of another block in F. What happens? (Think carefully, this is perhaps not completely obvious.)
The read succeeds fine. LBFS only checks lease at open: close-open consistency means that once you have the data it is yours. Does not matter if other people revoke lease or not.

9. Normal disks will guarantee that sector writes happen atomically (i.e., if the machine crashes while doing a sector write either the write happens completely or not at all). You want to port a RAID 5 system to run on disks that only write a single *byte* atomically. What problem does this create for crash recovery and what is a reasonable way to fix it? (Hint: think about interleaving.)
This is going to cause problems with fixing disks after crash. For atomic sectors you have either the old or new data at a sector level. For atomic bytes you can have sectors

with a mix of old and new data. While we can use checksums to detect when a sector is corrupted, we won't have the old data anymore. Thus, it could be doing a large write to a stripe on it, crash, come back and all the sectors involved were trashed because none completed. There are a couple plausible solutions. One is to just do byte-level striping. Another is to do an LFS thing where you never do update in place. Or you could force it to only write one disk per stripe group (sucks for performance though).

10. Give the one line change in the MapReduce word count program (in the appendix of the paper) that would allow it to run correctly when run without a combining step, but would produce incorrect output when run with combining. (And, of course, make sure you say why this change causes things to break.)

    Change: value += StringToInt(input-¿value()); to: value ++; This assumes that it gets the output of reduce, which is always a "1."

    This example was mentioned in class.