Stanford University
Computer Science Department
CS 240 Quiz 3
Spring 2003

June 7, 2003

This is an open-book exam. You have 50 minutes to answer all 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 - 2	
3 - 4	
5 - 6	
7	
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 all of the following questions and, in a sentence or two, say *why* your answer holds. Your answer should be concrete enough that it is readily distinguishable from someone who has jotted down a string of high-level buzzwords.

1. The google search paper claims that a lot of care has been spent on data structures minimizing disk seeks. Give two examples supporting this claim (besides the obvious idea of caching).
   Compression of various things (wordids using diffs, hitlists, dont' store offsets over 4K). Hash tables used so you don't have to do linear disk scans. The indices themselves. In memory tables (lexicon) to eliminate disk accesses.

2. The google file system paper claims that each chunk has an associated version number that is periodically incremented to detect stale replicas. Is it a problem if a replica goes down and comes back up between these increments? (Be very concrete as to why this does or does not cause problems.)
   It does not cause problems. The write attempt will fail until either (1) the lease expires or (2) the node comes back up, in which case we can safely go forward since the relevent state has not been modified.)

3. Write an MC extension that checks the rule "if a pointer is checked against null, do not dereference it." You may assume that the system will kill variables that are reassigned. Something like:

```
sm internal_null_checker {
 state decl any_pointer v;

 /* Initial start state: match any pointer
    compared to NULL in code, put it in a 'null'
    state on true path, ignore it on false path. */
 start:
    { (v == NULL) } ==> true=v.null, false=v.stop
  | { (v != NULL) } ==> true=v.stop, false=v.null
  ;
   /* Give an error if a pointer in the null state
      is dereferenced in code. */
 v.null:
   { *v } ==> { err("Dereferencing NULL ptr!"); }
   ;
}
```

4. Pick some useful optimization from one of the papers that we discussed this quarter (besides the superpage paper) and describe how to implement it within a virtual machine monitor. Be concrete as to any difficulties that come from doing it in the the VMM rather than in the OS itself.

Question 5: Yes, we have no RAID (20 points)

1. (5 points) Assume you have a simple RAID system: four data disks (A,B,C,D) and one dedicated parity disk (P) that holds the bitwise xor of the data disks ($P = A$ *xor* B *xor* C *xor* D).

   If the OS crashes while writing to the RAID system, what do you have to do after the crash to recover to a good state? What happens if a disk blows up during this recovery? (Assume that all redundant information lives on the parity disk and that you do not have a log.)
   You will have to recompute all the parity blocks. We took off two points if you said it mattered which order they were written out in. If a data disk fails while reconstructing then we cannot recover (these are two faults: invalid parity, and lost data).

2. (10 points) Why might a log structured file system (LFS) run substantially faster than an update-in-place file system such as FFS on a RAID system? Give two weak points of LFS: do you expect these to be worse or better on a large RAID system?
   RAID has problems with small writes since you may have to read the parity block and potentially the old data before doing the write. The best case for RAID are writes across and entire "stripe" — in the example, a write that would hit A,B,C,D and P. LFS is perfect for writing large chunks of data. LFS will probabaly get better in some ways. The large RAID has larger capacity, which may well make cleaning easier — more free capacity (less cleaning needed) and larger amounts of data, which allow better seperation. Also, LFS has trouble with sequential reads after contig writes — the parallel disk heads can eliminate many seeks that would be required (as long as the data is smeared across different disks).

3. (5 points) Jimbo likes your work and gives you an additional dedicated parity disk for the system above. Draw a picture with all six disks (two parity, four data) that shows how you'd organize your system so that it can survive *any* two disks blowing up.
   P1 = A xor B xor C and P2 = A xor B xor D.