Stanford University
Computer Science Department
CS 240 Quiz 3 With Answers
Spring 2004

June 2, 2004

This is an open-book exam. You have 50 minutes to answer eight out of ten 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 - 8	
9 - 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. Give an end-to-end argument for why, in the context of a distributed file system, that close-to-open consistency is better than sequential consistency (every read of data is guaranteed to return the value of the last write). Note: your answer should fit the argument template given as the literal end-to-end argument.

   Sequential consistency is expensive. Most applications do not need it. (Most are organized as producer-consumer (emacs produces a file, gcc consumes it), which is why you need close-open as a minimum.) Those that do need it, need other stuff as well (locking), so just providing sequential with nothing else is too weak.

2. Describe how using LFS on top of RAID could improve the RAID recovery process after a system crash (not a disk failure) as compared to a FFS-RAID system.

   FFS does not "know" where a crash happened. Thus, a RAID system will have to scan all data sectors in all disks to recompute parity. In contrast, LFS knows what data was recently written (these are the segments after the checkpoint), which means the RAID system just has to recompute parity for these.

3. Deviant: There are many "return-owner" routines that return ownership of a pointer, which the caller must track; losing this pointer is an error. The `malloc` routine is an example:

```
int *contrived(void) {
        int *p = malloc(sizeof *p);
        if(!p)
                return 0;   // not an error: p = null.
        if(foo() < 0)
                return 0;   // error: lost p!
        return p;           // not an error: returned ref.
}
```

   Sketch how to statistically infer which routines return owned pointers and flag errors when these pointers are lost. Assume that a pointer has been safely handled if it is (1) returned, (2) assigned to another

variable, (3) or passed to any function. (Note: only give a local, intra-procedural algorithm rather than global, interprocedural one.)

Assume all routines that return pointers are "return-owner." Count: callsites as population (n), places where a pointer is lost as an error (c). Z-rank errors based n and c.

4. Deviant: You are trying to write a deadlock detector. You see the following two routines. Give a belief-style intuition for what the potential deadlock is and why:

```
void A() {
        lock(x);
        unlock(y);
}
void B() {
        lock(y);
        unlock(x);
}
```

Either (1) one of the unlocks is incorrect (2) or y is locked on entrance to A and x is locked on entrance to B. In the latter case this means there is a circularity: (1) somewhere acquires y, then calls A, which acquires x and (2) somewhere else acquires x then calls B which acquires y. Deadlock.

5. Disco: "the execution of the operating system in general and critical section in particular is slower on disco, which increases the contention for semaphores and spinlocks." What is the intuition for why contention increases, and what experiment demonstrates it?

They implicitly define lock contention as: you are trying to acquire a lock that someone holds. If everything else is equal, then as a critical section gets longer and longer, this obviously gets more likely. Disco overhead will dilate critical sections, thereby increasing contention. Figure 7 shows this: contention for memlock kills performance. 4 points for the explanation, 1 point for identifying the experiment. -1 for not relating back to Disco's virtualization overheads somehow.

6. Disco, Figure 5: why does "kernel time" decrease when running on Disco? Figure 6: why does Irix data grow? Why does buffer cache stay the same size for the M bar?

   kernel time is getting transferred to disco: disco initializes pages (helps pmake kernel time), and does bulk of tlb miss handling. Irix data grows because it is private and written (note: most of the Irix data is pages that were never *read from disk and hence not even eligible for sharing). Buffer cache stays same because of sharing trick and all VMs running the same workload. 2 points for Fig. 5, 2 points for Fig. 6, and 1 point for the buffer cache.*

7. *Concisely* give three examples where systems have handled the worst and normal cases separately.

   (a) *ESX: uses ballooing to get back pages, then goes to forcible paging then to forcible swapping.*

   (b) *VMS uses swapping if do not fit.*

   (c) *Most file systems intentionally do a bad job on the worst case of "write shared files" and are optimized for the normal case of non-write shared.*

   (d) *If you consider crash recovery, there are lots of places. Checkpoints in LFS can be viewed as a way to bound the worst case (crash recovery) by doing something that you don't use in the normal case (no crash).*

   (e) *Leases explicitly seperate worst case (node down) from normal case (node responsive).*

8. You know the future. *Concisely* describe three *important* optimizations you could do (and how) in a realistic operating system *besides* predicting which cache entry to evict.

   (a) *LFS: What things die together.*

(b) *Livelock: whether you will get livelock if you handle a packet. Plausible: if a packet will cause a queue overflow. More plausible: what packet an application is for.*

(c) *Scheduling: what thread will cause other threads to block waiting for it.*

9. Several papers have claimed that locking should be separate from the file system. This is somewhat naive. Explain how you would have to modify the cache coherence scheme before locking could be useful.

   If you don't make caches consistent before a lock acquisition or after a lock release, then the locks are pretty useless since programs can arbitrary versions of the data, which will make it difficult to guarantee any invariant about the file. A reasonable minimum coherency scheme would be "release consistency" where you ensure caches are consistent when a lock is released.

10. Describe how to do Eraser-style detection of file race conditions in an NFS file server. In particular explain: (1) what the states in the Eraser state diagram correspond to and (2) what information you would need besides just the RPC arguments for `read` and `write`. Note: You will almost certainly need to augment the NFS protocol and you should use the Eraser definition of what a race is.

    You'll have to add locking. You probably want to track when they open, close file so can filter out when a given access cannot propagate forward (past close) or backward (past open).

    (a) *Virgin: just got written.*

    (b) *Exclusive: one process just reading writing file (emacs).*

    (c) *Shared-modified: get a write from another process.*

    (d) *Shared: read of another thread. If firest thread closed, then go to exclusive state.*