Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.824 Distributed System Engineering: Spring 2010

Quiz I Solutions

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to answer this quiz.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

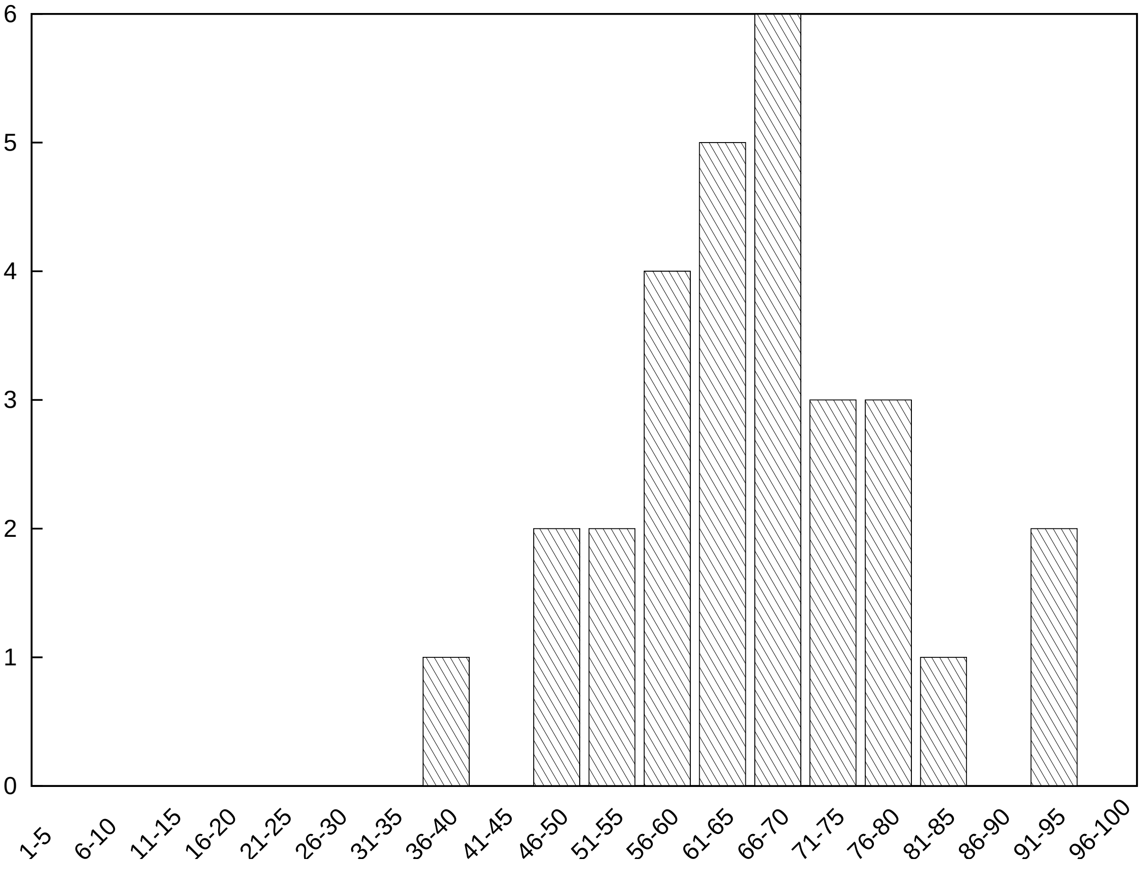
Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

I (xx/26)	II (xx/21)	III (xx/49)	IV (xx/4)	Total (xx/100)

Name:

Grade histogram for Quiz 1



max	=	92
median	=	66
μ	=	66.0
σ	=	12.2

I YFS

Working on Lab 4, Ben Bitdiddle adds a `create` function to his YFS client; the function is responsible for creating files. Ben wants to use multiple YFS clients so he also adds locking to his YFS client code. He uses a lock per inode and wraps each YFS client function with a call to the lock_client's `acquire` and `release` passing the inode's `inum` as the argument. Here are some excerpts from yfs_client.cc (error checking code omitted):

```
yfs_client::inum yfs_client::ilookup(inum di, std::string name)
{
  std::string buf;
  yfs_client::inum inum = 0;
  lc->acquire(di);
  ec->get(di, buf);
  ... // parse buf and set inum if file found
  lc->release(di);
  return inum;
}

int yfs_client::create(inum di, std::string name, inum ci)
{
  std::string buf;
  lc->acquire(di);
  lc->acquire(ci);
  ec->get(di, buf);
  ... // add (name, ci) pair to buf
  ec->put(di, buf);
  ec->put(ci, "");
  lc->release(di);
  lc->release(ci);
  return OK;
}
```

And from fuse.cc:

```
yfs_client::status fuseserver_createhelper(fuse_ino_t parent,
    const char *name, mode_t mode, struct fuse_entry_param *e)
{
  e->ino = yfs->ilookup(parent, name);
  if (e->ino == 0) {
    e->ino = random() | 0x80000000;
    yfs->create(parent, name, e->ino);
  }
  getattr(e->ino, e->attr);
  e->attr_timeout = 0.0;
  e->entry_timeout = 0.0;
  return OK;
}
```

Name:

1. [8 points]: To Ben's surprise his implementation fails the standard tests given by the 6.824 staff. Describe a test on which Ben's program will fail? How can Ben fix his code to ensure that his code passes the tests?

Answer: If multiple threads are concurrently creating the same file in the same directory then it is possible for `fuseserver_createhelper` to create multiple files with the same name in the directory. The following interleaving of instructions demonstrates the problem:

```
Thread 1                        Thread 2
e->ino = yfs->ilookup(parent,
                       name);
                                e->ino = yfs->ilookup(parent, name);
e->ino == 0 so execute:
yfs->create(parent, name,
                     e->ino);
                                e->ino == 0 so execute:
                                yfs->create(parent, name, e->ino);
```

Answer: To fix this, Ben needs to make the calls to `yfs_client::ilookup` and `yfs_client::create` atomic in `fuseserver_createhelper`. One way to do this is to create a `yfs_client::ilookup_wo` function that is exactly like `yfs_client::ilookup` but does not take out any locks and have `yfs_client::create` check, while holding the `di` lock, if the file exists using `yfs_client::ilookup_wo`.

In another attempt at correctness, Ben modifies his code in yfs_client.cc as follows:

```
yfs_client::inum yfs_client::ilookup(inum di, std::string name)
{
  std::string buf;
  yfs_client::inum inum = 0;
  lc->acquire(di);  // <------------------ B
  ec->get(di, buf);
  ... // parse buf and set inum if file found
  lc->release(di);
  return inum;
}

int yfs_client::create(inum di, std::string name, inum &ci)
{
  std::string buf;
  lc->acquire(di);  // <------------------ A
  ci = ilookup(di, name);
  if (ci == 0) {
    lc->acquire(ci);
    ci = random() | 0x80000000;
    ec->get(di, buf);
    ... // add (name, ci) pair to buf
    ec->put(di, buf);
    ec->put(ci, "");
    lc->release(ci);
  }
  lc->release(di);
  return OK;
}
```

And in fuse.cc:

```
yfs_client::status fuseserver_createhelper(fuse_ino_t parent,
    const char *name, mode_t mode, struct fuse_entry_param *e)
{
  yfs->create(parent, name, e->ino);
  getattr(e->ino, e->attr);
  e->attr_timeout = 0.0;
  e->entry_timeout = 0.0;
  return OK;
}
```

Name:

2. [6 points]: To his surprise his code still does not work; his code fails to create a file. What is the problem and how can he fix it?

Answer: The code will deadlock because a lock for `di` will be taken on the line labeled A and then the call to ilookup will take out the lock for `di` again on the line labeled B.
One way to fix this is to create a `yfs_client::ilookup_wo` function that is exactly like `yfs_client::ilookup` but does not take out any locks and have `yfs_client::create` call `yfs_client::ilookup_wo` instead of `yfs_client::ilookup`.

Now Ben wants to simplify YFS by expanding the extent protocol from `get`, `put`, and `remove` to `read`, `write`, `truncate`, `create`, `lookup`, etc. Every file system operation (i.e., each fuse operation) will be handled by the extent server. The YFS client will be a proxy for the extent server.

3. [6 points]: What are the advantages of executing the file system operations on the extent server? Which functions/features can he remove from YFS using this scheme?

Answer: Executing all the file system operations on one server means that there is no longer any need to have a lock server and the YFS clients do not have to worry about making calls to the lock server. The extent server can use Pthread mutexes to lock critical sections.

4. [6 points]: What are disadvantages of moving the file system operations out of the YFS client and into the extent server?

Answer: Since one single machine handles all file system requests from all clients, the system proposed by Ben will not be scalable. Even though the number of clients can grow, the processing capacity of the single extent server will remain constant. Ben's scheme does not exploit the processing capacity of the client machines to execute file operations.

Name:

II Short questions

5. [7 points]: An invocation o_1.f(o_2) on an RMI object o_1 takes another RMI object o_2 as an argument. The method f may invoke a method on o_2. The server executing f, however, may not have heard about o_2 before. How does Java RMI arrange that f can invoke a method on o_2? What information is in o_2's handle? Why is that enough for the runtime to do its job?

Answer: A reference to o_2 is passed to the server executing f. This reference contains the URI of o_2 which consists of the host machine identifier (IP address or host name) and the object ID. The URI allows the server executing f to uniquely name and find where o_2 resides. In addition, the JVM provides dynamic code loading which allows stubs to be dynamically loaded into the runtime; the server executing f need not know anything about o_2 prior to executing f.

6. [7 points]: Both TreadMarks and Tra use version vectors to maintain consistency, but Ivy as described in Section 3.1 of the paper doesn't. What in Ivy's design allows it to avoid using version vectors?

Answer: Ivy has a central server (the manager) that handles and serializes all requests and all data movement. For writes, the manager ensures that all readers of a pages respond to the invalidation request and arranges the movement of data to the writer before allowing the write to proceed. When the writer receives the data, it knows that it has received the most recent version and no checks requiring version vectors are necessary.

7. [7 points]: Bayou cannot guarantee the order in which writes are committed is consistent with the tentative order indicated by their timestamps. Using the calendar application, give a scenario with a few devices synchronizing with each other that shows the committed order being different from the tentative order.

Answer: Three devices (A, B, and C) need to be used to create such a scenario. All three devices choose to schedule an appointment for 10am with 11am and 12pm as alternatives. The conflict resolution procedure for all three devices is to choose 11am if 10am is not available and 12pm if 11am is not available. First, A synchronizes with B; the tentative order is A getting the 10am slot and B the 11am slot. Second, C synchronize with the primary and permanently gets the 10am slot. Finally, A synchronizes with the master and the final order will be C with the 10am slot, A with the 11am slot and B with the 12pm slot.

Name:

III Locking and consistency

Alice P. Hacker wants to understand consistency better and explores it with an application, namely a parallel program for solving the traveling salesman problem (TSP). She starts with the following sequential C program with a standard branch-and-bound algorithm:

```
// matrix that records for each town the distance "dis" to "totown"
struct dist {
        int totown;
        int dis;
} distance[NRTOWNS][NRTOWNS] =  {
#include "g.h"
};

int path[NRTOWNS];      // the path tsp is working on
int best_path[NRTOWNS];// the shortest path found so far
int min = 10000;        // length of the shortest path

// l is the number of cities on path
// len is the length of the path
tsp(l,len)
{
        int e,me,i;

        if (len >= min)    // prune?
                return;
        if (l == NRTOWNS-1) {  // found a shorter path?
                min = len;
                for (i = 0; i < NRTOWNS; i++) best_path[i] = path[i];
        } else {
                me = path[l];
                for (i=0; i < NRTOWNS; i++) {
                        e = distance[me][i].totown;
                        if (!present(e,l)) {  // if e is not on path, branch
                                path[l+1] = e;
                                tsp(l+1, len + distance[me][i].dis);
                        }
                }
        }
}
```

Alice runs the program by invoking `tsp(0,0)`, which computes shortest path starting with city 0 (i.e., `path[0]` is 0). In the first invocation of `tsp`, `tsp` extends the path through all neighbors of 0, and for each neighbor it invokes itself recursively. If the path has a length larger than the length of the best path found so far, an invocation returns and doesn't explore that path further, pruning the search.

Name:

She parallelizes the program for a cache-coherent shared-memory multiprocessor. The parallel program first creates many jobs, each job consists of a partial path with 4 different cities on it. Once the program is done creating jobs, each processor invokes the following procedure `worker`:

```
struct job {
  int path[NRTOWNS];
} jobs[NJOBS]
jobindex = 0;

worker() {
  int myj;
  while (jobindex < NJOBS) {
    myj = jobindex++;
    path = jobs[myj].path   // copy the job's partial path in my path
    tsp(4, len(path))
  }
}
```

Each worker repeatedly grabs the next uncompleted job and invokes `tsp` to try to extend the path to a complete path. Each processor has its own copy of `path`, but `jobs`, `jobindex`, `min`, and `best_path` are globally shared variables.

8. [7 points]: Rewrite the procedure `worker` with locks so that this program will work correctly.

Answer: Both the reading and the writing of `jobindex` must be protected. One way of doing that is as follows:

```
lock jobi_lock;

worker() {
  int myj;
  acquire(jobi_lock);
  while (jobindex < NJOBS) {
    myj = jobindex++;
    release(jobi_lock);
    path = jobs[myj].path   // copy the job's partial path in my path
    tsp(4, len(path))
    acquire(jobi_lock);
  }
  release(jobi_lock);
}
```

9. [9 points]: Annotate/modify the procedure `tsp` on the previous page with read/write locks for the shared variable `min` so that the parallel program will work correctly. (Read/write locks have an extra argument for `acquire` that says whether to grab the lock in read mode or write mode. If the lock is acquired in read mode, other processors can also acquire it in read mode, but a processor acquiring in write mode will be delayed until all readers have released their lock.)

Answer: The reading and writing of `min` must be protected, but care must be taken when writing `min`: `min` may only be written if the value being written is smaller than the original value of `min`. In other words, the check of `len >= min` and the write to `min` must be atomic. There are two ways to accomplish this: 1) upgrading a read-write lock; or 2) rechecking. A holder of a read-write lock in read mode can upgrade from read to write mode—a lock upgrade guarantees that no other thread is holding the same lock in read mode and that no other thread had the lock in write mode between the read and the upgrade. An alternative solution is to to recheck `min` before it is modified to make sure it is still smaller than `len`. The code for the upgrade solution is as follows:

```
acquire(min_lock, READ);
if (len >= min)     // prune?
        release(min_lock):
        return;
if (l == NRTOWNS-1) {  // found a shorter path?
        acquire(min_lock, WRITE); // This is the upgrade
        min = len;
        for (i = 0; i < NRTOWNS; i++) best_path[i] = path[i];
        release(min_lock);
} else {
        release(min_lock);
```

10. [5 points]: Why is it better to use read/write locks instead of locks that don't distinguish between read and write modes? Will the TSP program run faster or slower with read/write locks and why?

Answer: A read/write lock allows multiple threads to execute read-only critical sections in parallel; a regular lock would serialize the execution of all critical sections. TSP will run faster, because `tsp` reads `min` much more often than writing it, and those reads can now proceed in parallel.

Assume that you annotated all shared variables (including, for example, `best_path`) with locks and that the program functions correctly on a cache-coherent shared-memory multiprocessor.

Alice notices that the Athena clusters are seldom used and together they have many more free processors than her shared-memory multiprocessor. She installs TreadMarks and runs the parallel program on it. She runs TreadMarks with lazy release consistency.

11. [7 points]: Will the program compute the correct answer? (Briefly explain your answer.)

Answer: Yes, the program is appropriately labeled with locks. A properly labeled program on release-consistent memory executes with the same behavior as on a sequentially-consistent memory.

Name:

Alice observes that the performance on TreadMarks isn't great, because each invocation of the procedure `tsp` must take out a lock to read `min`. To reduce this overhead, she changes the program as follows. He modifies the program to use two `min`s, one per processor (`pmin`) and one shared (`min`, as before). Before a worker invokes `tsp`, it copies the shared `min` into the per processor `pmin`. She modifies `tsp` to read the per processor `pmin` at the pruning line. Also, when `tsp` finds a shorter path, he modifies `tsp` to read the shared `min` and if the shared `min` is larger than `pmin`, he updates the shared `min`.

12. [7 points]: Will the program compute the correct answer on TreadMarks? (Briefly explain your answer.)

Answer: Yes, even though the program is not properly labeled. A processor may update `min` while other processors may keep using `pmin`, which now contains a stale value. However, that doesn't result in incorrect behavior, because if a new path is found, a processor reads `min` again, and updates it only if `len` is indeed smaller than `min`. In addition, no path will be pruned too early, because `pmin` is always smaller or equal to `min`. So, the correct value of `min` will be computed.

13. [7 points]: Will the program with the `pmin` hack run faster than the program without the hack on TreadMarks? (Briefly explain your answer.)

Answer: Yes. A typical job will take out a lock only once, namely to read `min` at the beginning of the job and update `pmin`. So, no locks are acquired during a job and all communication associated with locks during a job is avoided. Only when `min` must be updated, will communication happen, but that will happen seldom, because the bound is strictly decreasing and will decrease quickly initially. A job may do a few extra recursions of `tsp`, but that should have a small effect on performance for three reasons: there are many jobs and each job starts with an up-to-date version of `min`; `min` changes infrequently (initially it will be updated a few times quickly, but then very infrequently); and communication in a network of workstation is very expensive (i.e., it stalls the computation for many cycles). The extra work is much less than the cycles saved by avoiding communication.

14. [7 points]: Alice also runs the `pmin` version on TreadMarks with eager release consistency. Will this version run faster or slower than with lazy release consistency? (Briefly explain your answer.)

Answer: The paper reports on this experiment and shows that eager is slightly faster (figure 9 shows that eager achieves a higher speedup). As described in Section 5.2, the TSP in the paper uses the same private minimum hack. The reason eager performs better is that 1) `min` is immediately updated on all processors when it is changed on one processor (thereby resulting in more immediate pruning of longer paths) and 2) the total communication is about the same for both eager and lazy (see figures 10, 11, and 12).

IV 6.824

15. [2 points]: Describe the most memorable error you have made so far in one of the labs. (Provide enough detail so that we can understand your answer.)

Most common answers:

concurrency
problems with FUSE
problems with the RPC library; RPC library does not type check arguments

We would like to hear your opinions about 6.824, so please answer the following two questions. (Any relevant answer will receive full credit!)

16. [1 points]: What is the best aspect of 6.824?

Most common answers:

papers
labs

17. [1 points]: What is the worst aspect of 6.824?

Most common answers:

FUSE
distributed debugging
questions at beginning of class
some papers are old
labs too open ended
quiz

End of Quiz I

Name: