Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.824 Distributed System Engineering: Spring 2009

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.

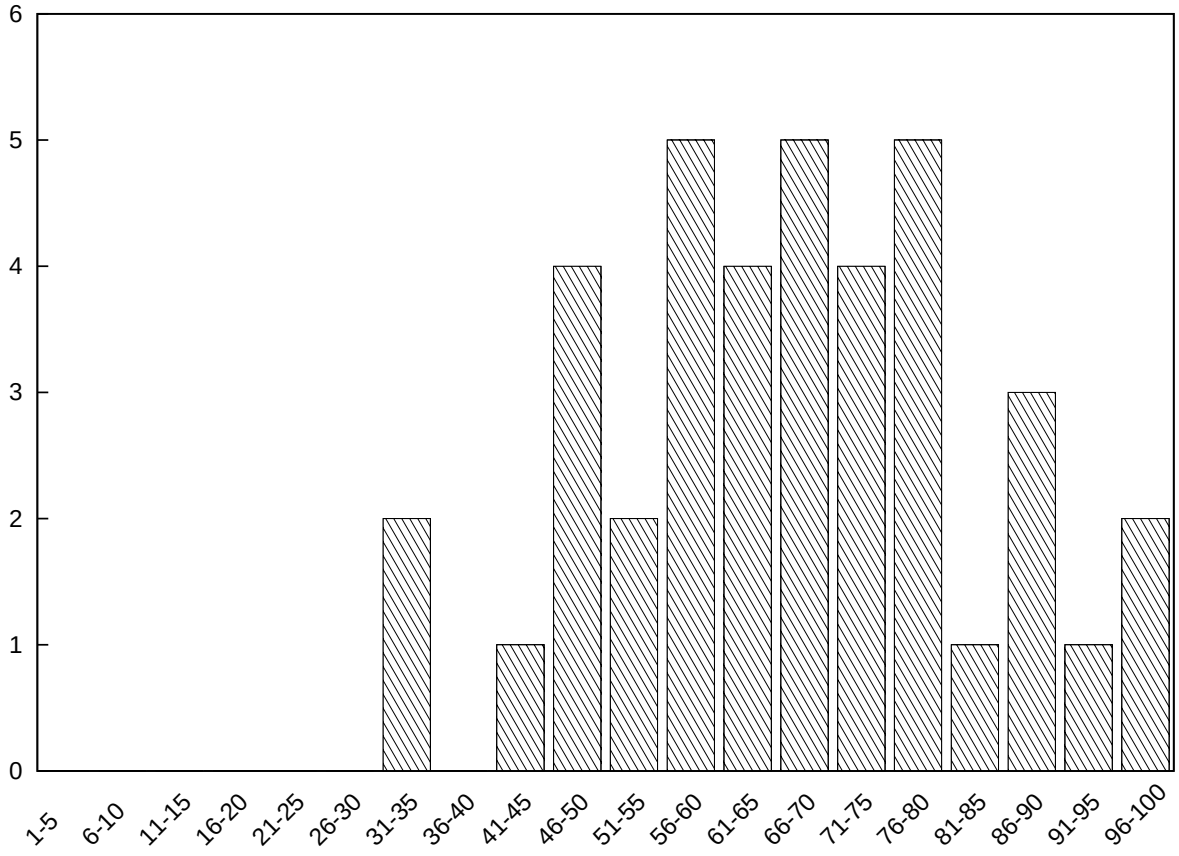
Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

I (xx/14)	II (xx/21)	III (xx/20)	IV (xx/21)	V (xx/14)	VI (xx/10)	Total (xx/100)

Name:

Grade histogram for Quiz 1



$$\begin{aligned} \text{max} &= 100 \\ \text{median} &= 67 \\ \mu &= 66.8 \\ \sigma &= 16.2 \end{aligned}$$

I Remote procedure call

Ben Bitdiddle makes the following changes to your RPC client from lab 1:

> Each time a request is retransmitted, he assigns it a new sequence number. His client remembers only the most recent sequence number assigned to each RPC, and if a reply with an unknown sequence number arrives, it is ignored. Additionally, the client automatically re-binds to the server if the server crashes.

Then Ben modifies your RPC server to ignore all RPCs with a sequence number less than or equal to the highest sequence number it received from the same client so far. For example, if it receives a request with sequence number 5 from client A, it will ignore any subsequent requests it receives from A with sequence numbers less than or equal to 5. Note that the RPC client's retransmission code increases the timeout with each retransmission, so RPCs will eventually complete if the server is up.

In the following problems, assume a single client with one thread, and no crashes. However, the network may drop, delay, duplicate, or reorder messages.

1. [7 points]: Recall that the SETATTR operation changes a file's length and updates the modification time, then returns the file's new attributes. Explain how the behavior of SETATTR could be different with the at-least-once RPC package the staff provided for Lab 1 versus Ben's RPC package.

The SETATTR RPC may be executed more than once with both packages. However, Ben's package guarantees that the value returned (and hence the modification time) corresponds to the last execution.

For example, suppose the client does a SETATTR followed by a GETATTR on a file. If no other clients modify the file, one might expect both calls to return the same modification time, which is what will happen with Ben's RPC package. With at-least-once RPC, however, SETATTR might return the modification time from the first of two executions of the call, in which case the subsequent GETATTR will return a different modification time.

2. [7 points]: Circle the FUSE operations below that can fail or produce incorrect results with Ben's RPC package, but work with your at-most-once RPC. Then underline one and describe what goes wrong with that operation.

LOOKUP READ WRITE MKDIR REMOVE

MKDIR and REMOVE might fail with Ben's package because it can cause duplicate requests to be executed. When the server processes a duplicate REMOVE request, for example, it sees that the file is already deleted and returns an error. LOOKUP, READ, and WRITE are idempotent and therefore don't have this problem.

Name:

II Short questions: Tra, Bayou, and DryadLINQ

3. [7 points]: On Monday, Alyssa and Ben fully synchronize their filesystems using Tra. On Tuesday, Alyssa makes some changes on her computer, and on Wednesday, Ben makes some changes on his computer. Then they fully synchronize again, and Tra reports no conflicts. Ben remarks that the results are different than they would be if they had performed the same operations on a shared YFS server (with sequential consistency). Give a simple example of operations Alyssa and Ben might have performed to lead to this situation, and briefly explain.

Here's a simple example:

Alyssa: `cp x y`
Ben: `cp y x`

Suppose `x` *originally contained* `"foo"` *and* `y` *originally contained* `"bar"`*. The result with sequential consistency is* `x="foo"` *and* `y="foo"`*, but the result with Tra is* `x="bar"` *and* `y="foo"`*.*

The key point is that Tra only tracks write-write conflicts, not read-write conflicts.

Some students found answers involving special cases with deletes, but it's more complicated to construct an example where Tra doesn't report a conflict.

4. [7 points]: Suppose that while David is disconnected from the network, he uses Bayou's meeting room scheduler application to reserve the Stata center playground for a 6.824 post-quiz party. MIT Daycare reserves the playground for the same time slot, and neither of them specify alternative times. Under what specific circumstances would David get a confirmed reservation for the playground?[1]

David gets a confirmed reservation if his update reaches the primary before any conflicting updates.

5. [7 points]: Section 3.3 of the DryadLINQ paper gives DryadLINQ's implementation of MapReduce. Name an optimization in Dryad that the original MapReduce doesn't have that might give Dryad better performance than MapReduce for the word frequency counting program you sketched for the reading response question. Briefly explain how the optimization helps for this program.

Valid answers include any of the dynamic optimizations performed by DryadLINQ, except for re-executing slow processes. (MapReduce mainly performs static optimizations.)

[1] P.S. Daycare won't let us use it—sorry.

III Mutexes and condition variables

Cy D. Fect has implemented a simplified implementation of the lock server from Lab 1. In Cy's version, there's a fixed table of 1024 locks; locks are never added or removed. Each lock has an associated condition variable `cv` and a mutex `mutex`. There is also a `global_cv` and a `global_mutex` for the entire lock server. Furthermore, `acquire()` and `release()` only take a single argument and don't return a value.

```
class lock_server {
  struct lock {
    pthread_cond_t cv;
    pthread_mutex_t mutex;
    bool held;
  } locks[1024];
  pthread_cond_t global_cv;
  pthread_mutex_t global_mutex;
 public:
  lock_server();        // initializes everything appropriately
  void acquire(lock_protocol::lockid_t lid);
  void release(lock_protocol::lockid_t lid);
}
```

Unfortunately, Cy is having trouble writing the `acquire()` and `release()` RPC handlers. For each of the following implementations, explain what problems can be caused by the incorrect use of mutexes or condition variables. If nothing can go wrong, simply write "CORRECT," and if there are multiple problems, describing one is enough. Ignore efficiency and issues like handling errors due to lack of memory; focus on the correctness of the synchronization.

The first one has been done for you to give you an example of the kind of answer we're looking for. To save you time, the parts that vary from one question to the next have been highlighted; the other lines of code are identical in all of the problems.

```
void lock_server::acquire(lock_protocol::lockid_t lid) {
    pthread_mutex_lock(&global_mutex);
    if (locks[lid].held)
        pthread_cond_wait(&global_cv, &global_mutex);
    locks[lid].held = true;
    pthread_mutex_unlock(&global_mutex);
}

void lock_server::release(lock_protocol::lockid_t lid) {
    pthread_mutex_lock(&global_mutex);
    locks[lid].held = false;
    pthread_cond_broadcast(&global_cv);
    pthread_mutex_unlock(&global_mutex);
}
```

ANSWER:

INCORRECT
Thread A acquires lock 1.
Threads B and C try to acquire 1 and wait on global_cv.
Thread A releases lock 1 and broadcasts.
B and C both wake up, and both now think they own the lock.

Name:

6. [5 points]:

```
void lock_server::acquire(lock_protocol::lockid_t lid) {
    pthread_mutex_lock(&global_mutex);
    while (locks[lid].held)
        pthread_cond_wait(&global_cv, &global_mutex);
    locks[lid].held = true;
    pthread_mutex_unlock(&global_mutex);
}

void lock_server::release(lock_protocol::lockid_t lid) {
    pthread_mutex_lock(&global_mutex);
    locks[lid].held = false;
    pthread_cond_signal(&global_cv);
    pthread_mutex_unlock(&global_mutex);
}
```

INCORRECT
Thread A waits for lock 1.
Thread B waits for lock 2.
Lock 1 is released, and thread B gets signaled, but sees lock 2 is still held and waits again.
Thread A continues to wait even though lock 1 is free.

7. [5 points]:

```
void lock_server::acquire(lock_protocol::lockid_t lid) {
    pthread_mutex_lock(&global_mutex);
    while (locks[lid].held)
        pthread_cond_wait(&global_cv, &global_mutex);
    locks[lid].held = true;
    pthread_mutex_unlock(&global_mutex);
}

void lock_server::release(lock_protocol::lockid_t lid) {
    pthread_mutex_lock(&global_mutex);
    locks[lid].held = false;
    pthread_mutex_unlock(&global_mutex);
    pthread_cond_broadcast(&global_cv);
}
```

CORRECT

It's fine to call `pthread_cond_broadcast()` after releasing the lock. This just means that any waiters will wake up slightly later; the important part is that the setting and checking of the `held` flag happens atomically. The example at the end of Section 5.3 of the threading paper does this, too.

Name:

8. [5 points]:

```
void lock_server::acquire(lock_protocol::lockid_t lid) {
    pthread_mutex_lock(&global_mutex);
    while (locks[lid].held)
        pthread_cond_wait(&locks[lid].cv, &global_mutex);
    locks[lid].held = true;
    pthread_mutex_unlock(&global_mutex);
}

void lock_server::release(lock_protocol::lockid_t lid) {
    pthread_mutex_lock(&global_mutex);
    locks[lid].held = false;
    pthread_cond_signal(&locks[lid].cv);
    pthread_mutex_unlock(&global_mutex);
}
```

CORRECT

This is similar to question 6, except that there is one condition variable per lock. It's okay to use `pthread_cond_signal()` *here because there's no chance of waking the "wrong" waiter.*

9. [5 points]:

```
void lock_server::acquire(lock_protocol::lockid_t lid) {
    pthread_mutex_lock(&global_mutex);
    pthread_mutex_lock(&locks[lid].mutex);
    while (locks[lid].held)
        pthread_cond_wait(&locks[lid].cv, &locks[lid].mutex);
    locks[lid].held = true;
    pthread_mutex_unlock(&locks[lid].mutex);
    pthread_mutex_unlock(&global_mutex);
}

void lock_server::release(lock_protocol::lockid_t lid) {
    pthread_mutex_lock(&global_mutex);
    locks[lid].held = false;
    pthread_cond_broadcast(&locks[lid].cv);
    pthread_mutex_unlock(&global_mutex);
}
```

INCORRECT
The thread calling `acquire()` *waits on* `locks[lid].cv` *while holding* `global_mutex`*, so other* `acquire()` *and* `release()` *operations are unable to proceed and the system deadlocks.*

Name:

IV Entry consistency

Alice hears about a new consistency mode, called entry consistency. If a lock l protects shared data D, then entry consistency is defined as follows:

- An acquire of l on processor p is not allowed to complete until all updates to D before the last release of l are visible at p.

10. [4 points]: Write some pseudocode for a simple program that would operate correctly with sequential consistency but incorrectly with entry consistency. (Briefly explain your answer.)

One example:

CPU 0:
```
foo = 42;
foo_is_set = true;
```

CPU 1:
```
while (!foo_is_set)
    ;
print foo;
```

Any program that is correctly annotated with locks will work with entry consistency, so you had to give an example that was not correctly annotated.

11. [4 points]: Give an example program that might induce less communication under entry consistency than lazy release consistency. (Briefly explain why.)

In the following example, we have to send the updates for both `x` and `y` from CPU 0 to CPU 1 in LRC, whereas in entry consistency, we only have to send the updates for `y`.

CPU 0:
```
acquire(lx);
x++;
release(lx);
acquire(ly);
y++;
release(ly);
```

CPU 1:
```
acquire(ly);
y++;
release(ly);
```

Name:

12. [8 points]: Alice implements a protocol for entry consistency. Show the messages that Alice's protocol might send when a program acquires a lock. (Briefly explain how the protocol differs from the one for lazy release consistency.)

The protocol is the same as LRC, except that there is a version vector per lock, and only updates associated with the lock being acquired are transferred.

It occurs to Alice that she could make a distinction in operations that update D and operations that only read D by using reader/writer locks. If two operations take out a read lock for the same critical section, then both operations can be executed in parallel. To take advantage of this, she defines reader/writer entry consistency. The parts that differ from plain entry consistency are underlined:

- Before a lock l in write mode is granted to processor p, no other processor may hold l.
- An acquire of l on processor p in read or write mode is not allowed to complete until all updates to D before the last release of l in write mode are visible at p.

13. [5 points]: Give a program that can induce less communication under reader/writer entry consistency than under plain entry consistency, where all locks are treated as write locks. (Briefly explain why this program introduces less communication.)

CPU 0:	*CPU 1:*
`acquire_write(lx);`	
`x++;`	
`release(lx);`	
	`acquire_read(lx);`
	`z = x;`
	`release(lx);`
`acquire_read(lx);`	
`print(x);`	
`release(lx);`	

This example can induce less communication in two ways. First, CPU 0 doesn't need to send a vector timestamp to CPU 1, since CPU 0 itself most recently released the lock in write mode. Second, CPU 1 doesn't have to send the new value of `z` *(which arguably should have been protected by a write lock anyway.)*

This optimization can be applied to LRC, too.

Name:

V Logging with checkpoints

14. [7 points]: In the Cedar system, the VAM updates are not recorded with write-ahead logging. How is the content of the VAM recovered? After recovery, does the VAM have an accurate list of all free disk blocks? (Briefly explain your answer.)

The VAM is reconstructed from the file name table, and is accurate after recovery. (Blocks referenced in the file name table are marked allocated, and all other blocks are marked free.)

15. [7 points]: In FSD, creating a file might involve the following sequence of events:

1. An application calls `create(`*`filename, file_contents`*`)`.
2. Cedar writes the data and leader pages to disk.
3. The `create` call returns `OK`.
4. Later, Cedar records the file name table changes in the log.
5. Cedar updates the file name table on disk.
6. The VAM is written to disk later when the system is idle.

Lem E. Tweakit thinks there is no need to write the data and leader pages synchronously, and modifies Cedar so that creates result in the following sequence of events:

1. An application calls `create(`*`filename, file_contents`*`)`.
2. The `create` call returns `OK`.
3. Cedar writes the data and leader pages to disk. **(changed)**
4. Later, Cedar records the file name table changes in the log.
5. Cedar updates the file name table on disk.
6. The VAM is written to disk later when the system is idle.

In Lem's version of FSD, this order is always followed, the log is flushed just as frequently as before, and everything else is the same. Give an example of what can go wrong with Lem's modified FSD that the original FSD handles correctly, or explain why Lem's changes cause no new problems. Don't worry about concurrency, operations other than `create`, hardware failures, or bugs.

Lem's changes don't cause problems. A potential concern is that `create` *returns* `OK` *before the data and leaders are written, so if the system crashes before Step 3 in Lem's version, the file won't be there after recovery. However, this can happen in the original FSD, too!*

The key insight is that Step 4, where the log record is written, is the commit point in both versions. That is, if there is a crash before Step 4, the file will not be present after recovery; if there is a crash after Step 4, then the file will be present after recovery. (Identifying where the commit point is is a good way to understand any system that provides atomic updates.)

Name:

VI 6.824

16. [4 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:

12 concurrency bugs (many of which were covered on the quiz!)
8 miscellaneous issues with STL
6 problems with FUSE

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

17. [3 points]: What is the best aspect of 6.824?

Most common answers:

31 labs
6 papers

18. [3 points]: What is the worst aspect of 6.824?

Most common answers:

11 issues with papers: too many (6), too old (2), other (3)
11 issues with labs: too open-ended (3), not enough comments (3), C++ / STL / RPC (3), other (2)
10 quiz

End of Quiz I

Name: