15-440 Distributed Systems
Midterm

Name:	
Andrew: ID	

October 12, 2010

- Please write your name and Andrew ID above before starting this exam.
- This exam has 10 pages, including this title page. Please confirm that all pages are present.
- This exam has a total of 100 points.

Question	Points	Score
1	18	
2	7	
3	15	
4	6	
5	25	
6	5	
7	6	
8	17	
9	1	
Total:	100	

A Short Answers

1. (18 points) True/False. Grading is +2 points for a correct answer and 0 points for either blank or incorrect. In other words, at the end of the exam, if you don't know the answer to any of these, *guess*, because you'll get more points in expectation. If you want to be nice to the course staff, mark the ones you guessed on with a "G" so that we have a better idea of what material to go over. We won't penalize you.

True, False After being awakened from `cond_wait` (condition variable wait), a process must immediately try to acquire the lock associated with the condition variable.

Solution: *False*: Upon awakening, the process is guaranteed to hold the lock.

True, False In addition to greatly reducing the overall failure probability, moving from a single hard drive to using RAID 1 (mirroring) can also increase the throughput of small random reads.

Solution: *True*: The small reads need only be sent to a single disk, so in theory, a RAID-1 system can double small random read throughput.

True, False RPC systems typically provide exactly-once semantics under arbitrary failures.

Solution: *False*: RPC systems *cannot* provide exactly-once semantics in all scenarios. They can either provide at-most-once or at-least-once.

True, False The TCP transport protocol provides a reliable, in-order bytestream abstraction.

Solution: *True*: Yes, yes it does.

True, False In general, two computers can be time synchronized to within half of the round-trip delay between them.

Solution: *True*: If you don't know more about the network between them, this is the accuracy limit. (Imagine a completely asymmetric network with almost no delay from A-B and almost all of its delay on the return trip from B-A.)

True, False Kernel threads (typically) allow processes to harness multiple CPUs at the same time.

Solution: *True*: They do indeed. This is one of the great things about kernel threads.

True, False Switching between two kernel threads is faster than switching between two user threads because it's done by the kernel.

Solution: *False*: It's usually faster to switch between two user threads because you don't have the overhead of switching into the kernel and back.

True, False A CPU must provide hardware support for atomic instructions in order for applications to be able to correctly synchronize.

Solution: *False*: It's expensive, but we can use the distributed mutual exclusion techniques from lecture 11 to provide mutual exclusion without hardware support. But most CPUs provide atomic instructions as a (very welcome) optimization and convenience for programmers.

True, False The compare-and-swap instruction can be used to implement atomic higher-level operations such as incrementing a variable without the need for locks.

Solution: *True.*

```
while (1) {
  int old_x = atomic_var;
  int new_x = old_x+1;
  if (compare_and_swap(atomic_var, old_x, new_x) == SUCCESS) {
    break;
  }
}
```

2. (7 points) Lamport clocks capture the "happens-before relationship" between two events. If L(e_1) < L(e_2), then we can say that e_1 happened before e_2. Can you use Lamport clocks to say whether e_2 happened *as a result of* e_1? Answer yes or no, and then give a brief (2 sentence) example that explains your answer.

Solution: No, Lamport clocks cannot capture causality. Although e_2 might have been determined to happen after e_1, we cannot say that e_2 was a response to the event e_1.

In the baseball analogy, assume that e_1 is the event that the batter leaves home plate, and e_2 is the event that the home plate umpire picks his nose. They happened at the same location (process), so we can sequence them absolutely using Lamport clocks, but we can't make any claim about whether the umpire picked his nose *because* the batter left. In reality, the two events were probably unrelated.

Distributed File Systems

3. (15 points) For each of the following situations, would you rather be using NFS or AFS? Give one *succinct* (1 sentence) reason why. Be specific.

   (a) A large number of users.

   Solution: AFS: Places less of a burden on the server. Scales better.

   (b) Watching only the first five minutes of a three hour video located on the distributed file system.

   Solution: NFS: Fetches blocks, so would not need to download the whole video.

   (c) Mounting /usr/bin where files are rarely, if ever, modified.

   Solution: AFS: Few or no callbacks instead of periodically checking for changes.

   (d) An important file (the 15-440 grades) is sometimes edited at nearly the same time (mere seconds apart) by the 15-440 staff members.

   Solution: AFS: Callbacks result in better consistency.

   (e) Congested network environment.

   Solution: AFS: Callbacks normally require fewer messages than periodically checking for changes.

4. (6 points) In Lab 2 you are implementing your own distributed file system using FUSE. List one advantage and one disadvantage to using FUSE for a distributed file system. Be succinct - 1 sentence per answer.

   Advantage:

   Solution: Works better on heterogenous systems since the file system code is run in user space instead of within the kernel. Does not require you to modify kernel code.

   Disadvantage:

   Solution: Not as fast as a file system implemented entirely within the kernel.

End of Days

5. (25 points) A. Generic Mellon wants to set up a calendaring system for some friends, where each friend has their own calendar. Only the owner of a calendar can modify their own calendar, but everybody can read everyone else's calendar. Multiple friends can read a calendar simultaneously, but a calendar can only be edited by one person at a time. A calendar cannot be read if it is being edited.

   Given the following data structures:

```
struct Calendar {
        int id;
        int owner;
}

Calendar * calendars;
```

   Please write the following functions to provide a locking mechanism for A.G.M.'s calendaring system. Feel free to use additional data structures (pseudocode is fine, syntax is not important), and to modify the Calendar struct as you wish. Keep in mind that these functions will be run by multiple threads at once. Please state any assumptions you make.

Solution:

```
struct Calendar {
        int id;
        int owner;
        pthread_cond_t rcond;
        pthread_cond_t wcond;
        pthread_mutex_t mutex;
        int num_readers = 0;
        int num_writers = 0;
}
```

Solution:

```
/*
 * This function gives access to reader (a person)
 * to calendar with id calendar_id and returns whether
 * reading was a success.
 */
bool read_calendar(int reader, int calendar_id) {
  Calendar * cal = calendars[calendar_id];
  pthread_mutex_lock(&(cal->mutex));
  while (cal->num_writers > 0){
    pthread_cond_wait(&(cal->rcond), &(cal->mutex));
  }
  cal->num_readers++;
  pthread_mutex_unlock(&(cal->mutex));
  return true;
}
```

Solution:

```
/*
 * This function is called by the reader
 * when he is done reading.
 */
void done_reading(int reader, int calendar_id) {
  Calendar * cal = calendars[calendar_id];
  pthread_mutex_lock(&(cal->mutex));
  cal->num_readers--;
  if (cal->num_readers == 0)
    pthread_cond_signal(&(cal->wcond));

  pthread_cond_signal(&(cal->rcond));
  pthread_mutex_unlock(&(cal->mutex));
}
```

Solution:

```
/*
 * This function gives access to calendar_id to writer
 * and returns whether writing was a success.
 */
bool write_calendar(int writer, int calendar_id) {
  Calendar * cal = calendars[calendar_id];
  if (cal->owner != writer) return false;

  pthread_mutex_lock(&(cal->mutex));
  while (cal->num_readers > 0 || cal->num_writers > 0){
    pthread_cond_wait(&(cal->wcond), &(cal->mutex));
  }
  cal->num_writers++;
  pthread_mutex_unlock(&(cal->mutex));
  return true;
}
```

Solution:

```
/*
 * This function is called by the writer when
 * he is done writing.
 */
void done_writing(int writer, int calendar_id) {
  Calendar * cal = calendars[calendar_id];
  pthread_mutex_lock(&(cal->mutex));
  cal->num_writers--;
  if (cal->num_writers == 0)
    pthread_cond_signal(&(cal->rcond));

  pthread_cond_signal(&(cal->wcond));
  pthread_mutex_unlock(&(cal->mutex));
}
```

A Very Exclusive Club

6. (5 points) You want to choose a way to provide mutual exclusion for the following scenario: You have 2 very high quality, fast computers both connected to the same Ethernet switch. Requests are sent to *both* computers, and only one should act upon the request. When the request arrives, it should be assigned to one or the other computer, which then processes it. Exactly one computer should handle each request.

   What mutual exclusion protocol would you use? Justify your answer in terms of its performance, implementation complexity, and other factors you feel are important. A few sentences will do.

   Solution:

   The best solution here is probably a token passing scheme. It's not robust to failures without some extra machinery, but our machines are pretty reliable. The latency is close to optimal: Whichever computer is holding the token when a request arrives services the request and then passes the token. (What's cool about this is that the mutex algorithm actually runs *in advance* of the request arriving; other schemes could, of course, be modified to do this as well - but here the engineering is easy.) The number of messages sent is small, only one per request.

7. (6 points) Instead of 2 computers, you now want to build the same system using 20 computers that you found in the "free computers" bin in Wean hall. These computers are cheap, pretty unreliable, and all have varying speeds. Don't worry about a computer crashing once a request was received, but now what mutual exclusion protocol would you use? Justify *numerically* in terms of the number of messages, the latency, reliability, etc., compared to the protocol you picked for the previous part:

   Solution:

   Eek, we better use something that's robust to failures. And with 20 computers, we probably don't want to risk setting up a linear token passing scheme because the latency will be too high. We'd probably start with a majority-based broadcast scheme based upon the bakery algorithm and see if the traffic is too high. It provides low latency (a few RTTs), it does send a lot of messages (20 per request), but it's robust. If that proves

8. (9 points) A friend who took 15-440 last year proposes a new protocol for you. He wants to get the efficiency of Maekawa's quorum system but with robustness to node failures. He calls it the *Cluster Mutex, Ultra*, and it works like this:

   Arrange the nodes in a grid, just like Maekawa's algorithm. Assume there are an exact power of two number of nodes for simplicity.

   `mutex_acquire`: The requesting node r picks *two* rows and *two* columns to send its requests to. (Recall that Maekawa just sent to one row and one column). It broadcasts a message to the $2\sqrt{N}$ nodes in its rows and the $2\sqrt{N}$ nodes in its columns, saying REQUEST(time, r).

   `recv REQUEST`: If node hasn't granted its VOTE to anyone yet, send VOTE(time, r) back to r. Otherwise it's voted for s, so send back SORRY(time, r, s).

   `recv VOTE`: When a node receives at least $\sqrt{N}$ column votes (from nodes in the columns it picked) and $\sqrt{N}$ row votes (from nodes in the rows it picked), it has the lock and can proceed.

   `mutex_release`: Broadcast RELEASE(r) to all of the row and column nodes in its quorum.

   (a) Explain why or why not this protocol provides each of the following properties of a distributed mutual exclusion protocol. Be succinct; you may refer to known behaviors of other protocols we discussed in class if you wish.

- Mutual exclusion

Solution: The protocol is broken: It doesn't provide mutual exclusion. Consider what happens in this case where two clients are trying to grab the lock. "o"s represent responses sent to client 1 and "."s represent responses sent to client 2:

o	.	o	.	o	.	o	.
.	o	.	o	.	o	.	o
o	.						
.	o						
o	.						
.	o						
o	.						
.	o						

Both clients receive an "OK" response from $\sqrt{N}$ people in a column and in a row, just as the protocol said – but they have a completely non-overlapping set, and so fail to achieve mutual exclusion.

- Fairness

Solution: The protocol is not fair. Even if it worked, it has the same fairness problem that the basic Maekawa protocol does: there is no lamport timestamping of the requests to ensure that they are sequenced by the time at the requestor.

- Bounded waiting

Solution: The protocol does not guarantee bounded waiting. It's possible that a node could have to wait forever because other nodes keep winning the lock. (There are no queues for keeping track of who wanted the lock, and so everyone has to contend when the lock is released. This contention is *likely* to be bounded in practice, but in theory, it's not.)

(b) (8 points) There is a very bad problem with this protocol that is specific to the *changes* from the basic Maekawa protocol (e.g., not one of the things you may have mentioned above that are known problems with Maekawa). Propose a fix by describing the changed parts of the protocol, and sketch a brief explanation (think "proof" but not formal) of why your revised protocol works.

Solution: Instead of requiring $\sqrt{N}$ from any row and any column, require $\sqrt{N}$ responses *all in the same row* (and all in the same column). This works because it turns into running two copies of the Maekawa algorithm using a permuted set of rendezvous nodes. Either one alone is enough to provide mutual exclusion properly. The protocol is less robust than your friend hoped, however: it can only handle a few failures.

Anonymous Feedback

9. (1 point) Tear this sheet off to receive one bonus point. We'd love it if you handed it in either at the end of the exam or, if time is lacking, to the course secretary.

   (a) Please list one thing you'd like to see improved in this class in the current or a future version.

   Solution:

   (b) Please list one good thing you'd like to make sure continues in the current or future versions of the class.