Nathan Nard (nnard2, 671610717) CS 425 – Homework 4 20181206

Problem 1:

a.) Conflicting pairs:

read(a, T1); write(a, baz, T2) → (T1, T2)
read(a, T1); write(a, baz, T2) (second pair) → (T1, T2)
read(c, T2); write(c, foo, T1) → (T2, T1)

Since the conflicting pairs are not all the same order this execution is not serially equivalent.

b.) Conflicting pairs:

read(a, T1); write(a, baz, T2) → (T1, T2)
write(a, baz, T2); read(a, T1) → (T2, T1)
read(c, T2); write(c, foo, T1) → (T2, T1)

Since the conflicting pairs are not all the same order this execution is not serially equivalent.

c.) Conflicting pairs:

read(a, T1); write(a,baz, T2) → (T1, T2)
read(a, T1); write(a,baz, T2)→ (T1, T2)
write(c, foo, T1); read(c, T2)→ (T1, T2)

Since the conflicting pairs are all the same order this execution is serially equivalent.

d.) Conflicting pairs:

read(a, T1); write(a,baz, T2) → (T1, T2)
read(a, T1); write(a,baz, T2)→ (T1, T2)
write(c, foo, T1); read(c, T2)→ (T1, T2)

Since the conflicting pairs are all the same order this execution is serially equivalent.

Problem 10:

a.) I do not agree with this design. While it may make programming easier server side, it does not permit practical applications in reality. By using the unix file system read/write-like semantics, you lose server side operation idempotence and statelessness. Without operation idempotence, you open up the DFS to concurrency issues and without statelessness it will make it harder for the DFS to recover when (not if!) failures occur. For these reasons her design choices are undesirable.

b.) Her design needs to be redone without unix file system read/write-like semantics. Doing so will attain operation idempotence and statelessness at the server side which will therefore permit concurrency and easier failure recovery for the DFS.

c.) You wouldn't read my answer anyways, so I'm not going to answer. (anywhere but facebook)

```
d.)           ok….
              |
          ):^/-<--<
```

Problem 8:
a.) Mark page as (w) → Do write.
b.) This is erroneous, P4 cannot be in write mode without being the owner and there can be no read mode processes when there’s a write mode process.
c.) Ask other processes to invalidate their copies of the page (Multicast) → Fetch all copies; use latest copy; mark it as (w); become the owner → Do write.
d.) This is erroneous, two processes cannot be holding the page in write mode at the same time. Only the owner may hold a copy of the page in write mode.
e.) This is erroneous, three processes cannot be holding the page in write mode at the same time. Only the owner may hold a copy of the page in write mode.
f.) Ask other processes to invalidate their copies of page (multicast) → mark page as (w), become owner → do write.
g.) Ask other processes to invalidate their copies of the page (Multicast) → Fetch all copies; use latest copy; mark it as (w); become the owner → Do write.
h.) This is erroneous, two processes cannot be holding the page in write mode at the same time. Only the owner may hold a copy of the page in write mode.
i.) **Assuming P1 is the owner**: Ask other processes to invalidate their copies of the page (Multicast) → Fetch all copies; use latest copy; mark it as (w); become the owner → Do write.
Assuming P1 is NOT the owner: This is erroneous, a process must be marked as the owner in order to hold a page in write mode.
j.) Ask other processes to invalidate their copies of the page (Multicast) → Fetch all copies; use latest copy; mark it as (w); become the owner → Do write.

Problem 5:

Kafka is a means to keep large volumes of incoming information and its processed output sorted by purpose or topic. Within the Kafka topology are many components, most notably are the Producers, Consumers, and the Kafka layer (a DFS) that separates the two. The Producers "publish" information in streams to the Kafka layer, which processes the incoming information and sorts the data into "topics", which are further divided into subtopics or "partitions", which are also streams. The data is appended to the partition streams and the "Consumers" read the partition streams that they are assigned or "subscribed" to as they are being written to by the Kafka layer.   The Consumers are divided into "consumer groups" which can be assigned to one "topic", but each consumer group member can be assigned to one or more partitions.

Source:
https://kafka.apache.org/intro

Problem 9:
a.) This scheme is "***<u>incorrect</u>***" (ill-advised). Hashing functions are "1-way" functions, meaning it will be virtually impossible to retrieve the original input from their outputs. Since the encrypted message is put through the hashing function, it will be impossible for anyone to determine the original encrypted message. Without knowing the encrypted message, those without Alice's public key will be unable to verify her identity. It would be possible if other's had Alice's private key (encrypt incoming message M using private key, then hash it and see if it equals the attached hash value), but handing out one's private key to others is not an advisable practice as there would be no confidence in who the original sender would be. A private key should only be owned by the original sender.

b.) This is similar to what I stated in the last scheme where multiple people had the private encryption key. In this case, everyone has a public encryption key. While it is possible to compare hash values of these encrypted messages as everyone has the public encryption key, there's no confidence in who the original sender is as anyone with the public encryption key could have sent it. So therefore it is "***<u>incorrect</u>***"

c.) This is again similar to the previous scheme. If Alice sends the encrypted hash of her message using her public encryption key, which multiple people may have, there's no way any receiver can be confident that Alice was the original sender as anyone can have the public encryption key. Therefore this is "***<u>incorrect</u>***"

d.) This is a ***<u>correct</u>*** method. Anyone with the public encryption key can get the hashed value of the message and compare it to their hashing of the message and be confident that Alice was the original sender because the encrypted hash was encrypted with Alice's private encryption key. Of course, this is going under the assumption that Alice is the only one with her personal private encryption key.

e.) This is a ***<u>correct</u>*** method, <u>although less efficient than scheme d</u>. Receivers with Alice's public encryption key will be able to decrypt the encrypted message and be confident that Alice was the original sender as she is the only one with her private encryption key. Compared to scheme d though, the decryption will result in the original message as opposed to a hash value of the message. As hashed values tend to be of constant, relatively smaller sizes, this can lead to larger encryptions to decrypt and therefore makes this method less efficient.

f.) Similar to schemes b and c, sending the message encrypted with the public encryption key is ill-advised as anyone with the public key can send that message. There's no confidence that Alice was the original sender. Therfore this is an "***<u>incorrect</u>***" method.

g.) As stated in parts d and e above, ***<u>scheme d is the most efficient and correct method</u>*** to verify Alice as the original sender of the message.

Problem 6:
ZAB and Paxos follow similar protocols with some noted differences. They both elect leaders, who propose values to followers and wait for a quorum of acknowledgments from the followers, and votes have timestamps. However ZAB primarily applies to synchronizing "incremental (delta) state updates" for primary-backup systems like Zookeeper while Paxos applies to general synchronization of "client requests" for state machine replication. When failures of a leader occurs, the new leader behaves differently in either algorithm. For Paxos, new leaders take overlapping concurrent client requests and arbitrarily decides the order to execute them, but this is not desirable for primary-backup systems; there must be a consistent ordering of overlapping requests when a new leader is elected. ZAB achieves consistent ordering of concurrent requests by including an extra synchronization phase during recovery and by "using a different numbering of instances based on zxids."

Source:
https://cwiki.apache.org/confluence/display/ZOOKEEPER/Zab+vs.+Paxos

Problem 2:

I am right. A proof by contradiction follows:

Let us assume we have a simple system of two transactions with two objects with different IDs and that this system is in deadlock. Let L1 and L2 represent the locks for the two objects and T1 and T2 be the two transactions in deadlock. In order for a deadlock to occur, a cycle of events like the following must occur:

T1 acquires L1 → T2 acquires L2 → T1 tries to acquire L2, but can’t due to T2 holding L2 → T2 tries to acquire L1, but can’t due to T1 holding L1 and hence deadlock.

By the specifications of the system, locks can only be acquired in descending lexicographic order of the respective object’s ID. Suppose the first object’s ID appears first in this ordering and that L1 goes with it, accordingly for the second object and the lock L2. In the simple deadlock scenario presented above, we see that T2 has acquired L2 before L1. This contradicts the system’s specifications and therefore this scenario could never happen. Therefore, with the system’s specifications, it is impossible to reach deadlock regardless of whether or not transactions acquire all of their locks at the start.□

Problem 3:

Example 1

T1	T2
Lock(A)	
Read(A)**	
Lock(B)	
Read(B)	
	Lock(C)
	Write(C)*
	Unlock(C)
Unlock(A)	
	Lock(A)
	Write(A)**
	Unlock(A)
Lock(C)	
Write(C)*	
Unlock(C)	
Unlock(B)	

Example 2

T1	T2
Lock(A)	
Read(A)	
Lock(B)	
Read(B)	
	Lock(C)
	Write(C)
	Lock(D)
	Read(D)
	Unlock(D)
Unlock(B)	
	Lock(A)
Lock(C)	

a.) The limited-greedy two phase locking does not satisfy serial equivalence. Example 1 Above is a counterexample. There are two conflicting operations with opposite orderings: The two Write(C)'s (*) have a (T2, T1) ordering while the Read(A) and Write(A) (**) have a (T1,T2) ordering. Therefore the example does not satisfy serial equivalence. The green cells are the "growing phase" and the red cells are the "shrinking phase". Notice how there's only one acquired lock per transaction in the red cells.

b.) Yes, like with traditional two phase locking, it will also be susceptible to deadlocking. Example 2 above is an example of a deadlock. Colors mean same thing as described in part a.