In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to answer this quiz.

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.

<table>
<thead>
<tr>
<th>I (xx/25)</th>
<th>II (xx/20)</th>
<th>III (xx/20)</th>
<th>IV (xx/20)</th>
<th>Total (xx/85)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name:

NYU ID:
I Multiple choice questions (25 points):

A. Which of the following statements are true about RPC?

1. Go’s RPC system provides at-most-once guarantee by saving every RPC request at the server durably on disk.
2. In order to guarantee at-most-once execution, the RPC server must only execute one request at a time.
3. In order to guarantee at-most-once execution, the RPC server must remember all RPC requests that it has executed and their corresponding replies.
4. When an at-most-once RPC call times out, the client cannot know whether the server has executed the request or not.

B. Which of the following statements are true about MapReduce?

1. When using MapReduce to run the “word-count” application over 1TB of data, the aggregate network traffic generated is \( \sim 1TB \) without the use of local combiner.
2. A mapper typically needs to fetch its input data from all machines in the MapReduce cluster before execution.
3. A reducer typically needs to fetch its input data from all machines in the MapReduce cluster before execution.
4. For a cluster of \( n \) machines, one should configure \( m \) mappers and \( r \) reducers such that \( m + r = n \) in order to maximize performance.

C. Which of the following things are true about a linearizability?

1. Both the Primary-backup system (e.g. lab2) and MultiPaxos can guarantee linearizability.
2. A linearizable system is also causally-consistent.
3. Suppose client C1 issues request \( \text{PUT}(x=1) \) before another client C2 issues request \( \text{PUT}(y=2) \). Assuming \( x \) and \( y \) have initial values zero and there are no other writes in the system. Then it’s not possible for client C3 to read \( y \)’s value to be 2 and then read \( x \)’s value to be 0.
4. It is the job of application programmers to ensure that a system is linearizable.

D. Which of the following things are true about causal consistency?

1. Causal consistency can not be realized in a scalable fashion.
2. Causal consistency offers weaker semantics than linearizability.
3. In a causally consistent storage system, data replication across nodes can be done in the background asynchronously.
4. In a causally consistent storage system such as Bayou, if a client C1’s write on \( x \) completes before another client C2 issues its read of \( x \), C2’s read is guaranteed to see C1’s write.
E. Which of the following things are true about transactions?

1. A database must execute transactions serially one after another in order to achieve serializability.
2. A transaction that only reads from the database does not need any concurrency control to ensure serializability.
3. Two phase locking requires locks to be acquired according to some total order (e.g. alphabetically according to lock names).
4. When using two-phase locking, a transaction $T_1$ may read the writes of a not-yet-committed transaction $T_2$.
5. Under two-phase locking, once some lock $x$ is released, then no locks can be acquired afterwards.
II Transactions (20 pts)

In the banking application, a customer can have both a “checking” and a “saving” account. The pseudocode below shows the implementation (WithdrawChecking and WithdrawSaving) for withdrawing money from either the checking and saving account, respectively. We assume there’s only one customer in the system. Data item “checking” stores the balance of the customer’s checking account and item “saving” stores the balance of the customer’s savings account.

WithdrawChecking(amount):
   c := db.GetInt("checking") //read the balance of "checking" as an integer
   s := db.GetInt("saving") //read the balance of "saving" as an integer
   c -= amount
   if c > 0 && c+s > 100 {//allow withdraw if there’s enough money in checking and combined account
      db.PutInt("checking", c) //write new balance to "checking"
      fmt.Printf("success, remaining=%d\n", c)
   } else if (c <= 0) {
      fmt.Printf("fail, not enough in checking %d\n", c)
   } else {
      fmt.Printf("fail, not enough in combined %d\n", c+s)
   }

WithdrawSaving(amount):
   c := db.GetInt("checking") //read the balance of "checking" as an integer
   s := db.GetInt("saving") //read the balance of "saving" as an integer
   s -= amount
   if s > 0 && c+s > 100 {
      db.PutInt("saving", s) //write new balance to "saving"
      fmt.Printf("success, remaining=%d\n", s)
   } else if (s <= 0) {
      fmt.Printf("fail, not enough in saving %d\n", s)
   } else {
      fmt.Printf("fail, not enough in combined %d\n", c+s)
   }

For each of the questions below, we assume these initial values for checking and saving data items: checking=100, saving=100.

1. [5 points]: Suppose WithdrawChecking is enclosed in a *serializable* transaction. Suppose one thread executes WithdrawChecking(40) while another thread concurrently executes WithdrawChecking(50). What are the potential printouts for each thread and the corresponding final checking balance after both operations finish? List every potential outcome in a row in the table below. (We assume WithdrawSaving is never invoked)
2. [5 points]: Suppose WithdrawChecking is *not* enclosed in a transaction. Suppose one thread executes WithdrawChecking(40) while another thread concurrently executes WithdrawChecking(50). What are the potential printouts for each thread and the corresponding final checking balance after both operations finish? List every potential outcome in a row in the table below. (We assume WithdrawSaving is never invoked)

<table>
<thead>
<tr>
<th>WithdrawChecking(40)’s printout</th>
<th>WithdrawChecking(50)’s printout</th>
<th>final value of checking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. [5 points]: Suppose `WithdrawChecking` and `WithdrawSaving` are each enclosed in a **serializable** transaction. Suppose one thread executes `WithdrawChecking(80)` while another thread concurrently executes `WithdrawSaving(90)`. What are the potential printouts for these two operations and the corresponding final values of `checking` and `saving` after both operations finish? List every potential outcome in a row in the table below.

<table>
<thead>
<tr>
<th>WithdrawChecking(80)'s printout</th>
<th>WithdrawSaving(90)'s printout</th>
<th>final value of checking</th>
<th>final value of saving</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. [5 points]: Suppose `WithdrawChecking` and `WithdrawSaving` are each enclosed in a transaction with **snapshot isolation** guarantee. Suppose one thread executes `WithdrawChecking(80)` while another thread concurrently executes `WithdrawSaving(90)`. What are the potential printouts for these two operations and the corresponding final values of `checking` and `saving` after both operations finish? List every potential outcome in a row in the table below.

<table>
<thead>
<tr>
<th>WithdrawChecking(80)'s printout</th>
<th>WithdrawSaving(90)'s printout</th>
<th>final value of checking</th>
<th>final value of saving</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
III  Paxos (20 pts)

Ben Bitdiddle wants to use Paxos to replicate a key-value store across multiple servers. Ben asks Alyssa P. Hacker to help him design the key-value store. In Alyssa's design, any server can process the clients' GET and PUT requests. More concretely, she gives Ben the following pseudocode for how each server should handle the "GET" and "PUT" request from the client. (For simplicity, the code below does not include mutex operations but assumes that access to any shared variable are correctly synchronized across multiple threads at the same server.)

In the pseudocode, Alyssa assumes a black-box Paxos implementation (line 4). Specifically, \( \text{op}' = \text{Paxos}(i, \text{op}) \) invokes Paxos to reach consensus among servers to decide on \( \text{op} \) for instance-\( i \). If this consensus is successful, this function returns \( \text{op}' = \text{op} \). Otherwise, it returns a chosen operation \( \text{op}' \) that is different from the suggested one \( \text{op} \).

```plaintext
RPCHandler(op):
for {  
i = pick any i such that log[i].status is not DECIDED nor DECIDING  
log[i].status = DECIDING  
op' = Paxos(i, op) //Paxos returns op' which has been decided for instance i  
log[i].op = op'  
log[i].status = DECIDED  
if op == op' {  
break  
}  
wait until for all i' < i, log[i'].status is DECIDED  
j = largest i'<=i such that log[i'].op.type=="PUT" and log[i'].op.key==op.key  
reply to client "OK", log[j].op.value  }
```

5. [5 points]: Alyssa tries to convince Ben that her pseudocode is "correct". Can you help Alyssa formalize what correctness means here? In other words, what is the consistency semantics that Alyssa tries to achieve in her design?
6. **[5 points]**: In the pseudocode above, is it possible that statement at line 11 can take an arbitrarily long time to finish? Please explain your answer.

7. **[5 points]**: Ben decides to delete line 11 since he cannot figure out whether or not it might cause a very long delay. Is Ben’s modification correct? If so, please explain. If not, please give an example “bad” execution. Your counterexample must be simple and involve no more than 2 concurrent operations.
In Alyssa’s pseudocode, both the \texttt{GET} and \texttt{PUT} operations invoke the Paxos protocol to associate each operation with a unique instance number. Ben tries to optimize the \texttt{GET} operation to bypass Paxos since \texttt{GET} is a read-only operation. In Ben’s design, he uses Alyssa’s RPC handler for processing \texttt{PUT}s and uses a different RPC handler to process \texttt{GET}s. His \texttt{GET} RPC handler works as follows:

\begin{verbatim}
GetRPCHandler(op):
  1: i = largest i such that log[i].status is DECIDING or DECIDED
  2: wait until for all i’ <= i, log[i’].status is DECIDED
  3: j = largest i’ <= i such that log[i’].op.type == "PUT" and log[i’].op.key = op.key
  4: reply to client "OK", log[j].op.value
\end{verbatim}

8. [5 points]: Is Ben’s modification correct? If so, please give a simple argument. If not, please explain with a counterexample.
IV Linearizability (20 pts)

Consider the following 4 histories of execution for a set of concurrent PUT and GET operations (assuming all keys have an initial value of 0). There are two clients in each history. The history shows the timeline of their operations. The black bar marks the beginning and end of the each operation.

(a) client-1:
client-2:
Put(x,1)
Get(x)=1
Get(x)=1
Put(x,2)
Get(x)=2
(b) client-1:
client-2:
Put(x,1)
Get(x)=1
Get(x)=1
Put(x,2)
Get(x)=2
(c) client-1:
client-2:
Put(x,1)
Get(x)=1
Get(x)=1
Put(x,2)
Get(x)=2
(d) client-1:
client-2:
Put(x,1)
Get(x)=1
Put(x,2)
Get(x)=2

Figure 1: Four example execution histories

9. [10 points]: Which of the above histories are linearizable? For each linearizable history, please give the equivalent serial execution schedule that produces the same execution results as the original history.
Ben Bitdiddle has a single-machine key-value store that stores and retrieves key-value tuples on disk. As a result, the system has long latencies and quite low throughputs. Ben reads online that one should use an in-memory cache system to accelerate a slow disk-based storage system. Therefore, he decides to run a high-performance single-machine in-memory key-value as a cache. The cache key-value service runs on a separate machine than the disk key-value service.

Ben writes a client library to process each client’s GET and PUT requests as follows. All the RPC calls below are synchronous in that the client program only proceeds after the corresponding RPC reply has been received.

**PUT(key, value):**
- cacheKV.Delete(key) //do RPC call to cacheKV service to delete the key
- diskKV.Put(key, value) //do RPC call to diskKV service to store the key-value tuple.

**GET(key, value):**
- v, ok := cacheKV.Get(key) //do RPC call to cacheKV service to get the key-value tuple
  - if ok {
      - return v, ok
  }
- v, ok = diskKV.Get(key) //do RPC call to diskKV service to get the key-value tuple.
  - if ok {
      - cacheKV.Put(key, v) //do RPC call to cacheKV to cache the retrieved tuple.
  }
  - return v, ok

10. **[10 points]:** Assume the diskKV service is linearizable and the cacheKV service is also linearizable. Does Ben’s above implementation of PUT and GET operations linearizable? If yes, please provide a brief argument for why. If not, which (if any) of the history in Figure 1 is possible in Ben’s implementation. Please describe the sequence of events that result in the non-linearizable history.