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/20)</th>
<th>II (xx/20)</th>
<th>III (xx/20)</th>
<th>IV (xx/20)</th>
<th>V (xx/10)</th>
<th>Total (xx/90)</th>
</tr>
</thead>
</table>

Name:
I  Multiple choice questions (20 points):

Answer the following multiple-choice questions. Circle all answers that apply. Each problem is worth 4 points. Each missing or wrong answer costs -2 point.

A. Which of the following statements are true about RPC system used in YFS?
1. Each RPC call is blocking.
2. The rpcc class only supports the execution of a single client thread.
3. If the rpcc library issues two RPC calls, x followed by y, to the same server, then the server must handle x before y.
4. The rpcs server creates a new thread to handle every incoming RPC request.
5. The RPC library uses UDP messages for communicating RPC requests and replies.

B. Which of the following statements are true for various consistency models?
1. Obeying all causal orderings is sufficient to guarantee state convergence.
2. Sequential consistency guarantees state convergence.
3. Sequential consistency guarantees causal ordering.
4. Sequential consistency does not admit a scalable implementation because it requires a total ordering of all operations in the system.

C. Which of the following things are true for Bayou and COPS?
1. Both Bayou and COPS guarantee causal consistency.
2. Causal consistency is sufficient to guarantee that an airline never overbooks its seats when multiple clients reserve seats on the same airplane using Bayou or COPS.
3. COPS authors choose not to support sequential consistency because sequential consistency is not scalable.
4. The design of Bayou requires each server to store the entire state of the system whilst each COPS server only needs to store a subset of state.

FatCat bank has the restriction that the combined checking balance \(C\) and saving balance \(S\) should never drop below zero. Transaction T1 attempts to withdraw 100 dollars from the checking account and transaction T2 attempts to withdraw 100 dollars from the saving account \(S\). Initially, the account balances are \(C=90\) and \(S=20\). The pseudocode of both transactions are as follows:

```
T1:
if (C+S > 100) {
    C -= 100
    print "success", C, S
} else 
    print "failure", C, S

T2:
if (C+S > 100) {
    S -= 100
    print "success", C, S
} else 
    print "failure", C, S
```
D. Which of the following outcomes are possible when FatCat bank uses a database supporting snapshot isolation?

1. T1 prints "success -10 20", T2 prints "failure -10 20".
2. T1 prints "success -10 20", T2 prints "success 90 -80"
3. T1 prints "success -10 -80", T2 prints "success 90 -80"
4. T1 prints "failure 90 -80", T2 prints "success 90 -80"
5. T1 prints "success -10 -80", T1 prints "success -10 -80"

E. Which of the following outcomes are possible when FatCat bank uses a database supporting degree 1 (read committed)?

1. T1 prints "success -10 20", T2 prints "failure -10 20".
2. T1 prints "success -10 20", T2 prints "success 90 -80"
3. T1 prints "success -10 -80", T2 prints "success 90 -80"
4. T1 prints "failure 90 -80", T2 prints "success 90 -80"
5. T1 prints "success -10 -80", T1 prints "success -10 -80"
II RPC

Ben Bitdiddle wants to implement a ping-pong service using the RPC library in yfs labs.

class PingPong {
  public:

  PingPong(int port, string dst) {
    pthread_mutex_init(&m, NULL);
    pings_sent = pongs_received = 0;
    srv = new rpcs(port);
    srv->reg(ben_protocol::ping, this, &PingPong::ping_handler);
    srv->reg(ben_protocol::pong, this, &PingPong::pong_handler);
    cl = new rpcc(dst);
    cl->bind();
  }

  void do_pingpong() {
    ScopedLock ml(&m);
    int r;
    cl->call(ben_protocol::ping, "ping", r);
    pings_sent++;
    printf("one round of ping-pong\n");
  }

  void ping_handler() {
    int r;
    cl->call(ben_protocol::pong, "pong", r);
  }

  void pong_handler() {
    ScopedLock ml(&m);
    pongs_received++;
  }

  private:
  rpcc *cl;
  rpcs *srv;
  pthread_mutex lock m;
  int pings_sent;
  int pongs_received;

  void main() {
    PingPong s = PingPong(atoi(argv[1]), string(argv[2]));
    while (1) {
      do_pingpong();
    }
  }
}

1. [10 points]: Ben launched two local processes to test his PingPong program.
Unfortunately, he noticed that his program is stuck with nothing being printed out on the screen. Please help Ben explain why his program behaves like this.

2. [10 points]: Help Ben fix the bug. You may directly write on top of Ben’s code.
III Crash Recovery

Ben Bitdiddle decides to make the lock server he built in Lab 1 fault tolerant. He logs every acquire and release operation to an on-disk file. The following pseudocode shows Ben’s modified acquire and release RPC handlers that include logging.

```c
int lock_server::acquire(int clt, lock_protocol::lockid_t lid, int &r)
{
    pthread_mutex_lock(&server_mutex);
    lock *l = locks[lid]; //assuming lock exists
    while (l->state==LOCKED)
        pthread_cond_wait(&(l->cond), &server_mutex);
    l->state = LOCKED;
    pthread_mutex_unlock(&server_mutex);
    log_operation(lid,LOCKED); //log_operation flushes the entry to disk
    return lock_protocol::OK;
}

int lock_server::release(int clt, lock_protocol::lockid_t lid, int &r)
{
    pthread_mutex_lock(&server_mutex);
    lock *l = locks[lid]; //assuming lock exists
    l->state = FREE;
    pthread_cond_broadcast(&(l->cond));
    pthread_mutex_unlock(&server_mutex);
    log_operation(lid,FREE); //log_operation flushes the entry to disk
    return lock_protocol::OK;
}
```

After the server reboots from a crash, it reads the list of logged operations and initializes the lock table state accordingly, as shown in the following pseudocode.

```c
//run before the rebooted lock server accepts any request
void recover()
{
    lock_protocol::lockid_t lid;
    lstate state;
    while (get_next_log_entry(&lid, &state)) {
        locks[lid] = new lock(lid);
        locks[lid]->state = state;
    }
}
```
3. **[10 points]**: Ben has observed empirically that sometimes there are two clients holding the same lock after recovering from a server reboot. Please give a concrete sequence of events that will lead to this scenario.

4. **[10 points]**: Please help Ben fix the bug in his code so that his lock server can recover correctly from a crash.
IV Consistency

Alyssa P. Hacker is running a social networking startup. She built her website on top of the PostgreSQL database, configured with the "serializable" isolation level. After the singer Psy has joined and posted music video “Gangnam Style”, Alyssa’s website has surged in popularity and the centralized database is now choking under the load. Alyssa has hired Ben Bitdiddle to scale her website.

Ben has decided to deploy a cluster of machines running an in-memory key-value store to cache the database queries collaboratively. The client API for in-memory key-value store includes get, put and remove. The key-value store supports sequential consistency.

Ben proceeds to modify the social networking application to use the distributed cache. One important feature of Alyssa’s site is to let a user follow other people that he likes. The following pseudocode (with syntax loosely based on PHP) contains the original functions to display the list of people followed by a given user and to insert a user’s follow link.

```php
function add_link(follower, followee)
{

    //insert the link into the graph table in centralized database
    database_query("INSERT into graph (from, to) VALUES ('$follower', '$followee')");
}

function obtain_followees(user)
{

    //obtain the user’s followers from the centralized database
    $result = database_query("SELECT * FROM graph WHERE from='$user'");

    return $result;
}
```

5. [10 points]: Please help Ben add the appropriate code to add and retrieve cached results in order to speed up displaying the list of people that a user follows.
6. [10 points]: Alyssa is feeling nervous about Ben’s design and the potential consistency problem that it introduces. However, Ben claims that the overall database system with his distributed cache still guarantees serializability since the cache itself is sequentially consistent. Is Ben right? If not, can you help Alyssa find a concrete example that violates serializability? (Hint: what happens when there are concurrent obtain/followees and add/link operations?)
7. [5 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.)

We would like to hear your opinions about the class so far, so please answer the following two questions.

8. [3 points]: What is the best aspect of this class?

9. [2 points]: What is the worst aspect of this class?

End of Quiz I