I Reading Questions

The following questions refer to Herbert Simon's paper, "The Architecture of Complexity" (reading #2).

1. [2 points]: Simon's notion of hierarchy organizes a collection of components by their

(Circle the BEST answer)

A. physical containment relationships

2/2

B. names and access patterns

[✓ **C.**] strengths of interaction

✓

2. [2 points]: Simon argues that systems naturally evolve in hierachical form because

(Circle the BEST answer)

[✓ **A.**] hierarchies are inherently stable

2/2

B. hierarchies are easily described

C. a component in a structure of size N can be accessed in $logN$ steps

✓

3. [6 points]: Based on the description of the X Window System in the 1986 paper by Scheifler and Gettys (reading #5), which of the following statements are true?

(Circle True or False for each choice.)

A. True / False X's client/server architecture ensures that one misbehaving client cannot interfere with other clients running on the same display.

Answer: False. For example, one client could use the window id of another client to write to its window.

0/2

B. True / False X's asynchronous protocol ensures that clients never have to wait for a network round-trip time for the server to respond to a request.

Answer: False. If the client needs some information from the server, such as a window id or the contents of a pixmap, it must wait for the server's reply.

C. [✓ **True**] / **False** The X protocol always requires the server to send an exposure event to the client to redraw its window when an obscured window becomes visible.

X

4. [6 points]: Based on the description of the UNIX file system in the 1974 paper by Ritchie and Thompson (reading #6), which of the following statements are true?

(Circle True or False for each choice.)

OMIT b/c answers were not removed from scans

A. True / False The kernel does not allow users to create hard links to existing directories.

Answer: False. The superuser may create a link to an existing directory, for example while creating the `..` entry for a sub-directory.

Note: We also accepted True for this answer, since it was perhaps unclear whether "users" included the superuser.

B. True / False An application can ask the kernel to read any file by specifying its i-node number, as long as that i-node represents a file that the application has permissions to read.

Answer: False. No system call takes an i-number as argument.

2/2

C. True / [✓ False] File names and i-node structures are stored within the data blocks of their containing directory.

✓

5. [6 points]: Based on the description of the MapReduce system in the 2004 paper by Dean and Ghemawat (reading #8), which of the following statements are true?

(Circle True or False for each choice.)

A. True / False If there are m map tasks, using more than m workers in the map phase may still improve performance beyond that achieved with m workers.

Answer: True. MapReduce will start multiple copies of the last few map or reduce tasks, to attempt to finish quickly despite slow or failed nodes.

B. True / False To achieve locality, map workers always execute on the same machine as the input data that they consume.

Answer: False. The master tries to place map workers with the input data, but if it can't it places them elsewhere (preferentially nearby in the network topology).

2/2

C. True / [✓ False] Intermediate data passed between the map workers and reduce workers is stored in the Google File System (GFS).

✓

6. [8 points]: The following question refers to the Eraser system, by Savage et al. (reading #7). Consider the following snippet of code, as part of a larger system:

```
Lock L1;
Lock L2;
int x;

function foo() {
  acquire(L1);
  print(x);
  release(L1);
}

function bar(int v) {
  acquire(L2);
  if (v == 0) {
    print(x);
  }
  release(L2);
}
```

The functions `foo()` and `bar()` are executed from separate threads, but Eraser never flags an error. Which of the following reasons might explain this?

(Circle ALL that apply)

X [✓ **A.**] `foo()` and `bar()` both execute, but never at the same time, so no race condition actually occurs

X **B.** `foo()` and `bar()` run concurrently, but `bar()` is always called with 1 as an argument

✓ [✓ **C.**] Every time `foo()` or `bar()` is called, an additional lock `L3` is also held

X **D.** The value of `x` is changed for the last time *before* either `foo()` or `bar()` are called for the first time.

2/8

II FaceFeeder

Inspired by Design Project 1, Ben BitDiddle decides to build a dataflow processing system for Facebook feeds. A *Facebook feed* is a stream of notifications, informing Facebook users of changes to a given friend's profile page.

Users upload programs, or *operators*, that are run by Ben's dataflow server. Operators can read from one or more feeds, and produce outputs which are themselves feeds. Users may subscribe to different feeds to receive alerts. Multiple users may subscribe to the same feed, and feed alerts are delivered asynchronously (e.g., via email.)

As an example of a FaceFeed application, one user might create an operator that combines their friends' "25 things you didn't know about me" lists into "a whole lot of things you didn't know about a whole lot of people" list. Another user might write an operator that takes in a stream of text and produces a graph showing the most common words in that stream. A third user might combine these together to produce a graph of the most common words used in his or her friends' "25 things you didn't know about me" list.

7. [8 points]: As a first approach, Ben decides to run all operators in the same address space, in a single process, with each operator running in a thread. His thread scheduler is pre-emptive, meaning that it can interrupt one thread and switch to another. Alyssa P. Hacker warns him that a single process is a bad idea, because running operators in the same process only provides soft modularity between them. Which of the following are problems that *could* arise in this design?

(Circle ALL that apply)

[✓ **A.**] One operator might corrupt the memory of another operator.

8/8

[✓ **B.**] One operator might produce improperly formatted results, and send them to another operator, causing that operator to crash.

[✓ **C.**] One operator might execute an illegal instruction, causing the process running the operators to crash.

D. One operator might never relinquish the CPU, preventing other operators from running.

8. [8 points]: Based on Alyssa's observation, Ben decides to switch to a new design where each operator runs in its own process, with its own address space, and with operators communicating only indirectly via the kernel. The OS handles scheduling of the processes, and is also pre-emptive. He claims this provides strong modularity and will prevent the problems Alyssa mentioned. Alyssa agrees that this will fix some of the problems with a single address space, but says it doesn't completely protect operators from each other. Which of the following are problems that *could* arise in this design?

(Circle ALL that apply)

A. One operator might corrupt the memory of another operator.

8/8

[✓ **B.**] One operator might produce improperly formatted results, and send them to another operator, causing that operator to crash.

C. One operator might execute an illegal instruction, causing the processes running other operators to crash.

D. One operator might never relinquish the CPU, preventing other operators from running.

Ben decides to continue with his one-process-per-operator desgin. Because operators aren't running in the same address space, Ben needs to use a kernel structure to exchange data between them.

Ben thinks that users of feeds will often be interested in the most recent updates first. Because of this, he decides to design the system so that upstream operators first send their newest items to downstream operators. To achieve this, he uses a stack abstraction rather than a queue like the bounded buffer we studied in class.

Ben decides to add two new routines to the kernel, `put_stack` and `get_stack`, which add an item to a stack and receive an item from a stack, respectively. Adjacent operators in the data flow graph exchange data by having the upstream operator call `put_stack` and the downstream operator call `get_stack`.

His implementation of these routines is listed on the following page.

```
// add message to stack, blocking if stack is full
// stack size is N
// initially head = N
// buffer is a 0-indexed array of N message slots
// stack.lock is a lock variable associated with the stack
put_stack(stack, message):
  while true:
    if stack.head > 0:

      acquire(stack.lock)
      stack.head = stack.head - 1
      release(stack.lock)

      acquire(stack.lock)
      stack.buffer[stack.head] = message
      release(stack.lock)

      return
    else
      yield() //let another process run


//get next message from stack, blocking if stack is empty
get_stack(stack):
  while true:
    if stack.head < N:

      acquire(stack.lock)
      message = stack.buffer[stack.head]
      release(stack.lock)

      acquire(stack.lock)
      stack.head = stack.head + 1
      release(stack.lock)

      return message
    else
      yield() //let another process run
```

Notice that Ben's implementations acquires and releases `stack.lock` several times in each function. Ben claims this improves the performance of his implementation (versus an approach that acquires the lock and holds it for the duration of several operations).

Suppose that two operators, o_1 and o_2 are exchanging data via a stack s, and they perform the following sequence of operations. Here, time advances with the vertical axis, so if one operation appears above another operation, it finishes executing before the other operation begins. If two operations appear on the same line, it means they execute concurrently, and that arbitrary interleavings of their operations are possible (except, of course, that two operations cannot both be inside a critical section protected by `stack.lock`.)

o_1	o_2
put_stack(s,m_1)	
put_stack(s,m_2)	m = get_stack(s)
put_stack(s,m_3)	

9. [14 points]: Assuming `N = 4` and `head = 4` initially, after the above sequence of operations run, which of the following are possible states of the stack and the value of the `m` variable resulting from the call to `get_stack` in o_2?

(Circle ALL that apply)

[✓ A.]

`m =` m_2

Stack
0: empty
1: empty
2: m_3
3: m_1

[✓ B.]

`m =` m_1

Stack
0: empty
1: empty
2: m_3
3: m_2

✓

[✓ C.]

`m =` empty

Stack
0: empty
1: empty
2: m_3
3: m_1

D.

`m =` m_2

Stack
0: empty
1: empty
2: m_3
3: m_2

14/14

put m1
put m2
head--
get
→ msg
msg
head++
put m3

put m1
put M2
get

10. [6 points]: After Ben implements FaceFeed, his users create a dataflow program that consists of a long pipeline of many single-input, single-output operators. Ben runs this program in FaceFeed on a single core machine and finds that the performance isn't good enough. He decides to switch to a multi-core machine, but finds that, even though the operating system is properly scheduling his operators on different cores, he doesn't get much of a parallel speedup on this new machine. Which of the following are possible explanations for this lack of speedup?

(Circle ALL that apply)

[✓ **A.**] One of the operators is much slower than the others, so its execution time dominates the total execution of the pipeline. ✓

6/6

B. All of the operators are about the same speed, so there is little opportunity for parallelism in the graph. ✓

C. One of the operators is much faster than the others, and those other operators dominate the execution time of the graph. ✓

III BeanBag.com

You're running BeanBag.com, a food delivery service. Your customers place orders over the Internet to your order server, using special client software that you supply to them. The server maintains, for each customer account, the list of items that the customer currently has on order. The order server communicates with a separate warehouse server that arranges for shipping of items.

Your order server supports three requests:

- `CHECK_ORDER(acct)`: given an account number, returns the list of items in that account's current order.
- `ADD_TO_ORDER(acct, item)`: add an item to the list of items an account has on order. Returns the item.
- `SHIP_ORDER(acct)`: directs the warehouse server to ship the account's current item list by truck to the customer. Returns the list of items that will be shipped.

Your order server is single threaded. The code for your server is given on the following page.

```
server():
  while true:
    request = RECEIVE_REQUEST()
    process_request(request)

process_request(request):
  if request.type == CHECK_ORDER:
    reply = process_check(request)
  else if request.type == ADD_TO_ORDER:
    reply = process_add(request)
  else if request.type == SHIP_ORDER:
    reply = process_ship(request)
  else
    reply = "error"
  SEND_REPLY(reply)

process_check(request):
  reply = orders[request.acct]
  return reply

process_add(request):
  orders[request.acct] = append(orders[request.acct], request.item)
  reply = "added " + request.item
  return reply

process_ship(request):
  otmp = orders[request.acct]
  orders[request.acct] = empty
  send an RPC to the warehouse server, asking for otmp to be shipped
  wait for reply from warehouse
  reply = "shipped " + otmp
  return reply
```

The order server has one CPU and keeps all its data in memory; it does not use a disk.

The initial version of the client software sends a request message to the order server when the customer clicks on the Check, Add, or Ship button, waits for a reply from the order server, and displays the reply to the customer. The client always waits for one operation to succeed before submitting the next one. Neither the client nor the order server does anything to deal with the fact that the network can lose messages.

11. [10 points]: The network between the client and order server turns out to be unreliable: it sometimes discards their messages. The network between the order server and the warehouse server is perfectly reliable. Which of the following problems might customers observe that could be caused by the network failing to deliver some messages between client and order server?

(Circle ALL that apply)

[✓ **A.**] The client software could wait forever for a reply from the order server. ✓

[✓ **B.**] The customer might click on the Ship button but BeanBag might never ship the order to the customer. ✓

[✓ **C.**] The customer might click on the Ship button, receive no reply from the order server, but still receive the items in the current order from BeanBag. ✓

10/10

D. The customer might be shipped two of an item that he or she Added only one of. ✓

12. [8 points]: You modify the client software to re-send a request every five seconds until it gets a reply from the order server. You make no modifications to the servers. Which of the following problems might your modification cause?

(Circle ALL that apply)

[✓ **A.**] The customer might be shipped two of an item that he or she Added only one of. ✓

5/8

B. The customer might Add some items to the order and get replies, then click on Ship, and get a reply with an empty item list. ✗

C. ✓ The customer might Add some items to the order and get replies, then click on Ship, get a reply with the correct item list, and then receive two distinct shipments, each with those items.

You decide that you need higher performance, so you convert the code to use a threading package, with pre-emptive scheduling. The order server starts up a new thread to serve each request. In order to avoid races your new code holds a lock when manipulating customer orders.

New or modified lines are in **bold**.

```
Lock lock;
server():
  while true:
    request = RECEIVE_REQUEST()
    create_thread(process_request, request)

process_request(request):
  if request.type == CHECK_ORDER:
    reply = process_check(request)
  else if request.type == ADD_TO_ORDER:
    reply = process_add(request)
  else if request.type == SHIP_ORDER:
    reply = process_ship(request)
  else
    reply = "error"
  SEND_REPLY(reply)
  exit_thread()

process_check(request):
  acquire(lock)
  reply = orders[request.acct]
  release(lock)
  return reply

process_add(request):
  acquire(lock)
  orders[request.acct] = append(orders[request.acct], request.item)
  reply = "added " + request.item
  release(lock)
  return reply

process_ship(request):
  acquire(lock)
  otmp = orders[request.acct]
  orders[request.acct] = empty
  release(lock)
  send an RPC to the warehouse server, asking for otmp to be shipped
  wait for reply from warehouse
  reply = "shipped " + otmp
  return reply
```

Your friend predicts that threading will not help performance, since your order server has only one CPU.

You measure the total throughput and per-request latency with the new and the old order server using a collection of machines running simulated clients. Each generates a sequence of `CHECK_ORDER`, `ADD_TO_ORDER`, and `SHIP_ORDER` requests, issuing a new one as soon as the server replies to the previous request. Each client machine generates requests for a different account. You measure throughput in total requests served per second. You should assume that it takes zero time to switch between threads, that the time to acquire a lock takes no time beyond waiting for the current holder (if any) to release it, and that function calls and returns take zero time.

13. [10 points]: It turns out your friend is not correct. Which of the following are true about the throughput of the threaded server, compared to the original server?

(Circle ALL that apply)

X **A.** The average per-request latency is lower.

✓ **B.** Simultaneous requests in the server have to wait for each other to release the lock, leading to lower throughput.

✓ **C.** A thread for one client can be in `process_check()` while a thread for another client is in `process_add()`, leading to higher total throughput.

✓ [✓ **D.**] A thread for one client can be in `process_check()` while a thread for another client is in `process_ship()`, leading to higher total throughput.

7/10

At your friend's insistence you replace the order server with a shared-memory multiprocessor, and you measure its performance as above. You find that performance has not increased significantly beyond your single-processor deployment.

14. [6 points]: You hope to increase performance still further on your new multiprocessor. Which of the following would have the greatest positive effect on performance, while preserving correctness? **(Circle the BEST answer)**

A. Have a separate lock for each request, so that, for example, `process_check()` calls `acquire(check_lock)` and `process_add()` calls `acquire(add_lock)`.

[✓ **B.**] Have a separate lock for each account, so that, for example, `process_check()` calls `acquire(locks[request.acct])`.

C. Delete all the acquires and releases.

6/6

D. Delete the acquires and releases, and add an `acquire(lock)` at the start of `process_request()`, and a `release(lock)` just before the `exit_thread()`.

End of Quiz I

I Reading Questions

1. [4 points]: Based on the description of the Witty worm in "Exploiting Underlying Structure for Detailed Reconstruction of an Internet-Scale Event", by Kumar, Paxson and Weaver (reading #18), which of the following are true?

(Circle True or False for each choice.)

2/4

A. ✓ **True / [✓ False]** Bugs in the worm's design made Witty's behavior harder to analyze.

B. True / [✓ False] To remain effective at detecting worms, it is important for network telescopes to keep X their IP address ranges secret.

2. [4 points]: Which of the following hints appear in Butler Lampson's "Hints for Computer System Design" paper (reading #20), possibly in different words? Mark each True if it appears in the paper, and False if it does not.

(Circle True or False for each choice.)

X **A. True / [✓ False]** Keep secrets in an implementation, hiding from clients aspects that might change.

2/4

✓ **B. True / [✓ False]** Implementations are more important than interfaces, because implementations determine performance.

X **C. [✓ True] / False** On coding: don't get it right, get it written; you can always fix it later.

✓ **D. True / [✓ False]** Keep caches small: when in doubt, flush it out.

3. [8 points]: Based on the paper "Why Cryptosystems Fail", by Ross Anderson (reading #17), which of the following are true?

(Circle True or False for each choice.)

A. [✓ **True**] / **False** The paper argues that the traditional threat model for cryptosystems is wrong. ✓

B. True / [✓ **False**] The paper argues that secure systems cannot be designed the same way safety critical systems are. ✓

8/8 **C. True** / [✓ **False**] ATM security breaches require that the thief determine both your account number and PIN. ✓

D. True / [✓ **False**] For one bank's ATM network to provide access for a user from another bank, both banks must know the PIN key corresponding to the user. ✓

4. [8 points]: Based on the description of System R in the paper "The Recovery Manager of the System R Database Manager" by Gray, McJones, et al. (reading #21), which of the following are true?

(Circle True or False for each choice.)

A. True / [✓ **False**] RAM buffering of disk I/O helps ensure atomicity. ✓

4/8 **B. True** / [✓ **False**] Shadow copies, without a log, are sufficient to ensure atomicity in the presence of concurrent transactions that both update the same file. ✓

C. [✓ **True**] / **False** A transaction is guaranteed to survive a crash once its log entry is written to memory. ✗

D. True / [✓ **False**] Uncommitted transactions may have issued writes *before* the last checkpoint. Therefore checkpoints may include incomplete transactions. ✗

5. [8 points]: Based on the paper "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns", by Pincus and Baker (reading #16), which of the following are true?

(Circle True or False for each choice.)

4/8

A. **True** / [✓ **False**] By not allowing writes outside of the bounds of objects, Java eliminates all risk of attacks based on stack smashing, assuming that the VM and any native libraries are bug free. ✗

B. [✓ **True**] / **False** Setting the permissions on the stack memory to prevent execution of code would foil attacks based on "return into libc". ✗

C. [✓ **True**] / **False** Making the stack begin at a memory location chosen randomly at runtime would foil the original stack smashing exploit. ✓

D. [✓ **True**] / **False** Using function pointers presents additional opportunities for arc injection. ✓

6. [8 points]: Based on the description of ObjectStore in the paper "The ObjectStore Database System" by Lamb, Landis, et al. (reading #23), state whether each of the following is true or false.

(Circle True or False for each choice.)

4/8

A. **True** / [✓ **False**] If an existing program, with its own implementation of lists and sets, wants to use ObjectStore to make its data persistent, it must switch to ObjectStore's list and set collections. ✓

B. **True** / [✓ **False**] ObjectStore needs to know the location of all pointers in all persistent data structures. ✗

C. **True** / [✓ **False**] The caching protocol assumes the programmer will obtain a lock before modifying a persistent object. ✓

D. [✓ **True**] / **False** The locking protocol always allows applications to execute concurrently, as long as they are not accessing the same object. ✗

7. [8 points]: Based on the description of Porcupine in the paper "Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service" by Saito, Bershad and Levy, state whether each of the following is true or false.

(Circle True or False for each choice.)

A. True / [✓ False] If all of the servers storing mailbox fragments for some user are down, the system will not be able to accept new mail for that user. ✓

B. True / [✓ False] Assume you have a large-scale Porcupine deployment, there are more concurrent users than servers, and all users have similar usage patterns. Storing more mailbox fragments for each user would reduce throughput. ✗

C. True / [✓ False] A user that fetches but does not delete their mail from a Porcupine server twice in a row can see different messages, even if no new messages are received. ✗

D. [✓ True] / False If one user's mailbox fragment list is causing too much load on one server, Porcupine can move just that user's mailbox fragment list to another server. ✗

2/8

II BLOP

Ben Bitdiddle is building a distributed gambling system called Ben's Land Of Poker (BLOP). In BLOP, users play hands of poker (cards) against each other. Each user is given two *private* cards that the other users can't see. Three additional *public* cards (which can be seen by all users) are revealed one-by-one. Users place bets in four rounds of betting, one after users receive their two cards, and one after each public card is revealed. Bets are in dollars, are > 0, and are not more than a user's remaining balance. At the end of the fourth round of betting, the user with the best hand (according to the rules of poker) wins.

Each user in BLOP has an account with a balance that is stored on one of BLOP's servers. Different users playing in a hand may have their accounts hosted on different servers. Between hands, users can add money to an account with a credit card. BLOP credits a user's account when that user wins a hand. BLOP withdraws from a user's account whenever the user places a bet. During a given hand, one server is appointed a *leader* that is responsible for running the hand: it draws the cards, transfers money from users' accounts to a central *pot* that contains the money bet so far, and sends data to the clients.

During a hand, clients talk only to the leader. The leader sends information about public and private cards to the clients, who connect from their own desktop machines, and also updates the balances of accounts stored on the disk of the non-leader servers (the *subordinates*) and the balance of the pot stored locally on the leader's disk.

The pseudocode used by the leader is as follows:

```
run_hand:
  // (1) beginning of hand
  curPot = 0
  write(pot,0)  // store value of pot on disk

  for each client c:
    // handMsg tells clients about their cards
    sendRPC handMsg(privateCards[c]) to c

  for (round in [0..3])
    // collectBets does one round of betting with clients,
    // returning each of their bets
    bets = collectBets(clients)

    for each client c:
      // servers is an array that stores the subordinate
      //    server for each client's account data
      sendRPC deductMsg(c, bets[c]) to servers[c]
      curPot = curPot + bets[c]
      write (pot, curPot)  // update value of pot on disk
      if (round != 3) // last round is just for betting
         sendRPC handMsg(publicCards[round]) to c

  winner = computeWinner(hands)
  sendRPC deductMsg(winner, -curPot) to servers[winner]
  // (2) end of hand
```

The code to process deductMsg on each of the servers looks as follows:

```
deductMsg(account, amt):
  prevBal = read(account)
  if (prevBal > amt):
     write(account, prevBal - amt)
  else
     write(account, 0)
```

Assume that the network uses a reliable (exactly once) RPC protocol to between the leader and the subordinate servers, such that the leader waits to receive an acknowledgment to each sendRPC request before proceeding. Also assume that write operations are atomic—that is, they either complete or do not complete, and after they complete, balances are on disk.

Initially, Ben's servers run a hand without using any transactions, logging, or special fault tolerance. If the leader does not receive an acknowledgment to an RPC within two minutes, it tells the clients the hand is aborted but takes no other recovery action. If the leader crashes, the clients eventually detect this and notify the users that the hand has aborted. Initially, the leader performs no special action to recover after a crash.

During a hand, each client is given up to two minutes to place a bet. If they do not respond within two minutes (either because they left the hand, or their machine crashed), they forfeit the hand and lose any money they may have bet (play continues for the other clients in the hand.)

8. [6 points]: Which of the following could go wrong if one of the subordinate servers crashes in the middle of a hand, assuming only one hand runs at a time:

(Circle True or False for each choice.)

A. True / [✓ False] After the leader aborts the hand, and the failed subordinate restarts, it is possible for the sum of all of the on-disk balances of the users in the hand to be greater than when the hand started.

✓

B. [✓ True] / False After the leader aborts the hand, and the failed subordinate restarts, it is possible for the sum of all of the on-disk balances of the users in the hand to be less than when the hand started.

6/6

Alyssa P. Hacker tells Ben that he should use transactions and two-phase commit in his implementation of BLOP. He modifies BLOP so that reads and writes of the pot and of user accounts on the subordinates are done as a part of a transaction coordinated with two-phase commit and logs. All log writes go directly to an on-disk log. Ben's scheme operates as follows:

- Prior to beginning a hand (before the comment labeled (1)), the leader writes a start of transaction (SOT) log entry and sends each subordinate a BEGIN message. Each subordinate logs an SOT log entry.
- Prior to any update to the pot, the leader writes an UPDATE log entry. Prior to any update to a user account balance, subordinates write an UPDATE log entry.
- At the end of a hand (at the comment labeled (2)), the leader sends each subordinate a PREPARE message for the transaction. If the subordinate is participating in the transaction, it logs a PREPARED log entry and sends a YES vote to the leader. If the subordinate is not participating in the transaction (because, for example, it crashed and aborted the transaction before preparing), it sends a NO vote.
- If all subordinates vote YES, the leader logs a COMMIT log entry and sends a COMMIT message to each of the subordinates. Subordinates log a COMMIT record and send an ACK message.
- Otherwise, the leader logs an ABORT log entry and sends a ABORT message to each of the subordinates. Subordinates log an ABORT record, roll back the transaction, and send an ACK message.

Assume Alyssa's additions to Ben's code correctly implement two-phase commit, and that the system uses the standard two-phase commit and log-based recovery protocols for handling and detecting both leader and subordinate failures and recovery. Both two-phase commit and log-based recovery were discussed in lecture. Two-phase commit is described in Section 9.6.3 of the course notes, and log-based recovery is described in Section 9.3.3 and 9.3.4 of the course notes.

Ben also modifies his implementation so that if one of the subordinates doesn't respond to a `deductMsg` RPC, the leader initiates transaction abort.

9. [9 points]:

Which of the following statements about the fault tolerance properties of Ben's BLOP system with two-phase commit are true?

(Circle True or False for each choice.)

A. True / [✓ False] If a subordinate crashes after the leader has logged a COMMIT, and then the subordinate completes recovery, and the leader notifies all subordinates of the outcome of the transaction, it is possible for the sum of all of the balances of the users in the hand to be less than when the hand started. ✓

B. [✓ True] / False If the leader crashes before it has logged a COMMIT and then completes recovery and notifies all subordinates of the outcome of the transaction, the sum of all of the balances of the users in the hand is guaranteed to be equal to the sum of their balances when the hand started. ✓

9/9

C. True / [✓ False] If the leader crashes after one the subordinates has logged a PREPARE, it is OK for that non-leader to commit the transaction, since the transaction must have completed on the subordinate. ✓

10. [4 points]: Ben runs his system with 2 subordinates and 1 separate leader. Suppose that the mean time to failure of a subordinate in Ben's system is 1000 minutes, and the time for a subordinate to recover is 1 minute, and that failures of nodes are independent. Assuming that each hand uses both subordinates, and that the leader doesn't fail, the availability of Ben's system is approximately:

(Circle the BEST answer)

[✓ **A.** 499/500] ✓ $(.999)^2$

B. 999/1000

C. 999/2000

D. 1/1000

4/4

To increase the fault-tolerance of the system, Ben decides to add replication, where there are are two replicas of each subordinate.

Ben's friend Dana Bass suggests an implementation where one replica of each subordinate is appointed the *master*. The leader sends messages only to masters, and each master sends the balance of any accounts it hosts that were updated in a transaction to the other *worker* replica, after it receives the COMMIT message for that transaction. Masters do not wait for an acknowledgment from their worker before beginning to process the next transaction.

When a master fails, its worker can take over for it, becoming the master. When the failed replica recovers, it simply copies the balance of all bank accounts from the new master and becomes the worker. Dana's implementation does nothing special to deal with the case where a COMMIT completes on a master and the master fails before sending the transaction to the worker, which can result in the worker taking over without learning about the most recent committed transaction.

11. [9 points]: Which of the following statements about this approach are true, assuming that failures of masters and workers are independent, and that the leader node never fails:

(Circle True or False for each choice.)

A. [✓ **True**] / **False** Dana's approach improves the availability (that is, the probability that some subordinate responds to `deductMsg` for a given client's account) versus a non-replicated system, as long as the worker node can take over for a failed master in less than the time it takes for the master to restart. ✓

B. True / [✓ **False**] Dana's implementation ensures single-copy serializability, since a user can never see results of hands that reveal that the system is replicated. ✓

C. True / [✓ **False**] Suppose Dana modifies her approach to have three replicas for each subordinate (two workers and a master.) Compared to the the approach with two replicas per subordinate, this three node approach has lower availability since the probability that one of the three replicas crashes is higher than the probability that one of two replicas crashes in the original version. ✓

9/9

III BitPot

In order to back up your laptop's files, you sign up with BitPot. BitPot is an Internet-based storage service. They offer an RPC interface through which you can read and write named files. BitPot gives each of their customers an identification number (cid, an integer). The RPC interface looks like:

```
putfile(cid, filename, content)
getfile(cid, filename) -> content
```

`putfile()` and `getfile()` send their arguments over a network connection to the BitPot server, and wait for a reply.

BitPot provides a separate file namespace for each `cid`; for example, `getfile(1, "x")` and `getfile(2, "x")` will retrieve different data. Neither BitPot nor `getfile()` / `putfile()` do anything special to provide security. Here is what the BitPot server's RPC handlers do:

```
putfile_handler(cid, filename, content):
  name1 = "/customers/" + cid + "/" + filename
  write content to file name1 on the BitPot server's disk
  return a success indication

getfile_handler(cid, filename):
  name1 = "/customers/" + cid + "/" + filename
  if file name1 exists on the BitPot server's disk:
    content = read file name1
    return content
  else:
    return a failure indication
```

You are worried about the security of your files: that other people (perhaps even malicious BitPot employees) might be able to read or modify your backup files without your permission.

For all of the following questions, attackers have limited powers, including only the following:

- Observe any packet traveling through the network;
- Modify any packet traveling through the network;
- Send a packet with any content, including copies (perhaps modified) of packets observed on the network;
- Perform limited amounts of computation (but not enough to break cryptographic primitives);
- Read and write the contents of the BitPot server's disk (for attackers that are BitPot employees);
- Observe or modify the behavior of the BitPot server's software (for attackers that are BitPot employees);

Attackers have no powers not listed above. For example, an attacker cannot guess a cryptographic key; cannot guess the content of the files on your laptop; cannot observe or modify computations on your laptop; and cannot exploit buffer overruns or other bugs on your laptop or BitPot's servers (such as manipulating path names used to read and write files).

You should assume that there are no failures (except to the extent that the attacker's powers allow the attacker to do things that might be construed as failures).

Scheme One

You decide to encrypt each file you send to BitPot with a key that only you know, using a shared-secret cipher (see section 11.4.2 "Properties of `ENCRYPT` and `DECRYPT`" in the course notes). When you want to back up a file to BitPot, you call `backup1()`:

```
backup1(filename):
  plaintext = read contents of filename´from your laptop's disk
  ciphertext = ENCRYPT(plaintext, K)
  putfile(cid, filename, ciphertext)
```

and when you need to retrieve a file from BitPot, you call `retrieve1()`:

```
retrieve1(filename):
  ciphertext = getfile(cid, filename)
  plaintext = DECRYPT(ciphertext, K)
  print plaintext
```

`cid` is your BitPot customer ID and `K` is your cipher key.

Only you and your laptop know `K`. `ENCRYPT` and `DECRYPT` withstand all the attacks mentioned in 11.4.2.

12. [8 points]: Which of the following are true about Scheme One?

(Circle True or False for each choice.)

A. True / [✓ **False**] `retrieve1(f)` will return exactly the same data that your most recent completed call to `backup1(f)` for the same `f` read from your laptop's disk, despite anything an attacker might do.

B. [✓ **True**] / **False** Eavesdroppers watching packets on the network may see ciphertext but are very unlikely to be able to figure out the plaintext content of your files.

C. [✓ **True**] / **False** BitPot's employees may see ciphertext but are very unlikely to be able to figure out the plaintext content of your files.

D. [✓ **True**] / **False** If someone modifies one of the files BitPot stores for you, `retrieve1()` is guaranteed to print random data (or to signal an error).

6/8

Scheme Two

Your friend Belyssa says you need to use authentication, using SIGN and VERIFY as described in section 11.3.4 of the course notes. She's not sure quite how best to do this, and suggests the following plan.

Belyssa's plan operates at the RPC layer, beneath putfile() / getfile(). The client (your laptop) and the server (BitPot) each have a shared-secret signing key (Kc and Ks, respectively). Each SIGNs each RPC message it sends, and VERIFYs each RPC message it receives. Each of them ignores any received message that doesn't verify. For simplicity, assume there is only one client and only one server, so that the server doesn't have to manage a table of per-client keys. The client and server both know both Kc and Ks.

```
// client putfile() and getfile() send requests like this:
send_request(msg):
  T = SIGN(msg, Kc)
  send {msg, T} to BitPot

// the server calls this with each incoming network message:
receive_request(msg, T):
  if VERIFY(msg, T, Kc) == ACCEPT:
    result = call putfile_handler() or getfile_handler()
    send_reply(result)
  else:
    // ignore the request

send_reply(msg):
  T = SIGN(msg, Ks)
  send {msg, T} to client

// the client calls this with each incoming network message:
receive_reply(msg, T):
  if VERIFY(msg, T, Ks) == ACCEPT:
    process msg (i.e. tell getfile() or putfile() about the reply)
  else:
    // ignore the reply
```

Only your laptop and BitPot's server know `Kc` and `Ks`. `SIGN` and `VERIFY` withstand all the attacks mentioned in 11.3.4.

You execute the following procedure on your laptop using Scheme Two:

```
test():
  write "aaaa" to file xx
  backup1(xx)
  write "bbbb" to file yy
  backup1(yy)
  write "cccc" to file yy
  backup1(yy)
  retrieve1(yy)
```

That is, you back up file `xx`, then you back up two different versions of `yy`, then you retrieve `yy`. `test()` will print a value (from the call to `retrieve1()`).

13. [7 points]: Which of the following values is it possible for `test` to print?
(Circle ALL that apply)

A. `aaaa` ✗

[✓ **B.**] `bbbb` ✓ 5/7

[✓ **C.**] `cccc` ✓

D. `(^.^) insert witty message here (^.^)` ✓

Scheme Three

Your other friend, Allen, is uneasy about the properties of Belyssa's scheme. He proposes eliminating Belyssa's changes, and instead SIGNing and VERIFYing only in the backup and retrieve procedures.

Allen's new backup (called backup2()) includes SIGN's authentication tag in the "content" it sends to BitPot, and Allen's new retrieve extracts and VERIFYs the tag from the data sent by BitPot.

```
backup2(filename):
  plaintext = read contents of filename from your laptop's disk
  ciphertext = ENCRYPT(plaintext, K)
  T = SIGN(ciphertext, Kx)
  what = {ciphertext, T}
  putfile(cid, filename, what)

retrieve2(filename):
  what = getfile(cid, filename)
  {ciphertext, T} = what
  if VERIFY(ciphertext, T, Kx) == ACCEPT:
    plaintext = DECRYPT(ciphertext, K)
    print plaintext
  else:
    // ignore the getfile reply
```

K is the shared-secret cipher key from Scheme One, which only you and your laptop know. Kx is a shared-secret signing key known only to you and your laptop.

You execute the following procedure on your laptop using Scheme Three. (This is the same procedure as for the previous question, but uses backup2() and retrieve2()).

```
test2():
  write "aaaa" to file xx
  backup2(xx)
  write "bbbb" to file yy
  backup2(yy)
  write "cccc" to file yy
  backup2(yy)
  retrieve2(yy)
```

14. [7 points]: Which of the following values is it possible for `test2` to print?
(Circle ALL that apply)

[✓ **A.**] `aaaa`

[✓ **B.**] `bbbb`

✓ 7/7

[✓ **C.**] `cccc`

D. `(^.^) insert witty message here (^.^)`

IV Systems Design Experience

There are known knowns.
There are things we know
that we know.

There are known unknowns.
That is to say
There are things that we now know
we don't know.

But there are also unknown unknowns,
There are things we do not know
we don't know.

15. [2 points]: The aforementioned quote is highly applicable to the design of large computer systems. According to the guest lecture on May 11, who first uttered this sage advice?
(Circle the BEST answer)

A. Albert Einstein

B. Butler Lampson

C. Mahatma Gandhi

[✓ **D.**] Donald Rumsfeld

2/2

End of Quiz III

I Reading Questions

1. [8 points]: Based on the Unison paper entitled "How to Build a File Synchronizer", by Trevor Jim et al., state whether each of the following is true or false.

(Circle True or False for each choice.)

2/8

A. [✓ **True**] / **False** File synchronization in Unison is idempotent, meaning that, in the absence of failures or intervening modifications, running it once leaves the file system on both machines in the same state as running it twice.

✓

B. [✓ **True**] / **False** Synchronization in Unison is atomic, so that either all or none of the files are updated.

X

C. [✓ **True**] / **False** The Unix command `touch` changes a file's modification time without altering its contents. If a file is touched first in one replica and then in another, a subsequent attempt at synchronization will report a conflict.

X

D. [✓ **True**] / **False** Unison maintains an archive file to avoid scanning the file system directory structure when propagating changes.

X

2. [8 points]: Based on the description of LFS in the paper "The Design and Implementation of a Log-Structured File System" by Mendel Rosenblum and John K. Ousterhout, state whether each of the following is true or false.

(Circle True or False for each choice.)

A. **True / [✓ False]** Relative to the UNIX file system, LFS performs better on random read workloads. ✓

B. **True / [✓ False]** Performance of LFS is generally better when the file system is using a larger fraction of the total disk space, since reads are more likely to be sequential. ✓

C. **[✓ True] / False** Any disk writes made after the most recent checkpoint will be discarded during recovery. ✗

D. **True / [✓ False]** Suppose you write two large files (larger than available RAM) to disk, one written sequentially and one in random order. LFS will perform equally well when reading each of these files sequentially. ✓

6/8

3. [8 points]: Based on the description of RAID in the paper "A Case for Redundant Arrays of Inexpensive Disks (RAID)" by David Patterson et al., state whether each of the following is true or false.

(Circle True or False for each choice.)

A. [✓ **True**] / **False** Assuming the disk provides no error detection or correction, a RAID 1 controller can detect that it has read a corrupt block from disk.

X

B. **True** / [✓ **False**] ✓ A dedicated parity disk (RAID 4) increases throughput over RAID with distributed parity (RAID 5), by removing the overhead of parity-updates from other disks.

4/8 **C.** [✓ **True**] / **False** ✓ A five-disk RAID 4 array has a lower expected availability than a four-disk RAID 4 array (assume that, in each configuration, there is a single parity disk and each disk is identical—i.e., of the same size, type, age, manufacturer, etc.).

D. [✓ **True**] / **False** Assuming the disk provides no error detection or correction, a RAID 5 controller can correct errors in a corrupt block read from disk.

X

4. [8 points]: Assuming that BGP works as described in the "Wide-Area Internet Routing" paper by Hari Balakrishnan, state whether each of the following is true or false.

(Circle True or False for each choice.)

A. True / [✓ False] An autonomous system (AS) will commonly announce the same routes to its upstream providers and to its peers. X

B. [✓ True] / False MIT's routers, which speak BGP to their upstream providers, must have a default route to send packets to the rest of the Internet. X

4/8

For the following two questions, assume that routes to MIT at all routers in the Internet have converged, that there are no failures or policy changes, and that BGP MEDs are not used by any AS.

C. True / [✓ False] Suppose that MIT has *two distinct* upstream ISPs, both of whom advertise routes on behalf of MIT. It is possible for packets sent from a given remote AS to traverse different autonomous systems in their path to MIT. ✓

D. [✓ True] / False Suppose that MIT has *exactly one* upstream ISP, which advertises routes on behalf of MIT. It is possible for packets sent from some *AS X to MIT* to traverse different autonomous systems than packets sent from *MIT to X*. ✓

5. [8 points]: Based on the "TCP Congestion Control with a Misbehaving Receiver" paper by Stefan Savage et al, state whether each of the following is true or false.

(Circle True or False for each choice.)

A. **True** / [✓ **False**] ✓ Fixing ACK division by only accepting ACKs for packet boundaries prevents communication with a TCP Daytona stack that acknowledges intermediate bytes in a packet.

B. [✓ **True**] / **False** ✗ TCP Daytona's implementation of optimistic ACKing asks the sender to retransmit packets that were lost in transmission after being optimistically ACKed.

4/8

C. **True** / [✓ **False**] ✗ Optimistic ACKing can be mitigated by a sender without modifying receivers.

D. [✓ **True**] / **False** ✓ Fixing DupACK spoofing with nonces requires the sender to remember nonce values for at most one window size worth of packets.

6. [6 points]: This question refers to the description of NFS "Case study: The Network File System (NFS)" (section 4.5 in the course notes). In an attempt to improve the performance of his NFS server, Ben Bitdiddle modifies the NFS protocol implementation at the server to immediately respond to `WRITE` RPC requests, rather than waiting until the disk operation succeeds. Which of the following statements about Ben's new implementation are true, relative to the unmodified version?:

(Circle True or False for each choice.)

A. [✓ **True**] / **False** Latency of `read()` system calls on the client may be lower.

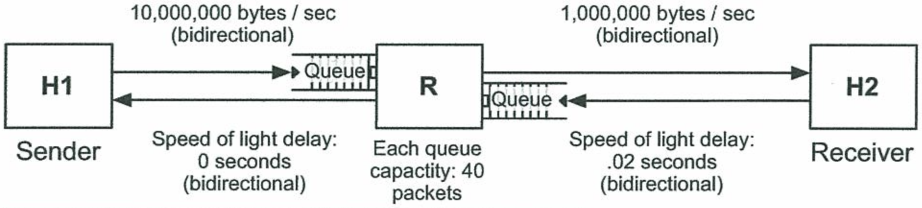
B. [✓ **True**] / **False** Latency of `write()` system calls on the client may be lower.

2/6

C. **True** / [✓ **False**] Latency of `close()` system calls on the client may be lower.

II Sliding Window

Ben Bitdiddle needs to transfer multi-gigabyte files from his radio telescope in California to his computer at MIT. He gets a special deal from Speedy Sam's Network Company, who supplies him with network-layer service between California and MIT on a private network (not part of the Internet). Speedy Sam's network topology looks like this:



Sliding window size:	10 packets
Data packet size size:	1,000 bytes
Acknowledgement packet size:	40 bytes
Sender retransmit timeout:	5 seconds

Figure 1: Ben's network configuration, with initial network parameters.

H1 is Ben's host in California; R is a router in the same room as H1; H2 is Ben's host at MIT. Both links are bi-directional, and the two directions operate independently. The H1—R link has a speed-of-light delay of zero, and a capacity of 10,000,000 bytes/second in each direction. The R—H2 link has a speed-of-light delay of 0.020 seconds, and a capacity of 1,000,000 bytes/second in each direction. Router R has two packet queues, one for each direction. When a packet arrives on a link, R adds it to the end of the queue feeding the other link. Each queue has maximum length of forty packets: if a packet arrives and the relevant queue already has 40 packets in it, R discards the packet. You can assume that the CPUs on H1, H2, and the router are infinitely fast (thus they do not impose any delays due to computation).

Speedy Sam's network carries IP packets. Ben's computers have IP-layer software but don't come with any transport-layer software, so Ben decides to design his own transport protocol.

All data packets are 1,000 bytes, including IP and link-layer headers. Ben's protocol splits the file to be sent into a sequence of segments, each of which fits in a packet, and numbers the segments sequentially (these are sequence numbers). Each data packet header contains the sequence number of the segment of data it contains; the sequence number field has enough bits to fit the largest possible sequence number. Acknowledgment (ACK) packets are 40 bytes long.

Ben's protocol uses a sliding window with a fixed window size of 10 packets. To cope with the possibility of lost packets, the sender re-sends each segment in the window every five seconds until it gets an ACK covering that segment from the receiver. When the sender receives an ACK that advances the window by n segments, it sends the next n segments as fast as the sender's link to the router allows. The receiver sends an ACK for each data packet it receives. Each ACK contains the sequence number of the lowest-numbered segment that the receiver has **not** received (i.e. ACKs are cumulative). Whenever an as-yet-unseen segment arrives at the receiver, the receiver hands the segment to the application (the destination part of the file transfer program); the receiver does not give the application duplicate segments.

7. [6 points]: At what approximate rate (in segments per second) will Ben's protocol deliver a multi-gigabyte file from H1 to H2?

(Circle the BEST answer)

A. 1000

[✓ **B.** 250]

C. 40

D. 25

E. 10

segment = 1000 bytes

10,000 → ← on router

1000 → ← on reciever

6/6 ✓

40

$\frac{10}{.04} = 250$

.04 seconds

$\frac{40}{.04} = 1000$

8. [6 points]: If Ben wanted to double the rate at which the system delivers file data from H1 to H2, what should he do?

(Circle the BEST answer)

A. Double the capacity of the H1—R link, to 20,000,000 bytes/second.

B. Double the capacity of the R—H2 link, to 2,000,000 bytes/second.

C. Double the maximum queue length in the router, to 80 packets. ✓

[✓ **D.** Double the window size, to 20 packets.]

E. Double the speed-of-light delay of the R—H2 link, to 0.040 seconds.

6/6

Name: Solutions

After a few months Ben's budget is cut, and he decides to save money by renting a lower-speed network from Speedy Sam. Sam reduces the capacity of the R—H2 link to 1,000 bytes/second (i.e. just one packet per second).

9. [6 points]: Ben starts a file transfer. His protocol sends out the first window of ten segments of the file. How long will it take from the start of the transfer until the sender receives an ACK for the last segment in that window?

(Circle the BEST answer)

A. 0.040 seconds

B. 1.020 seconds

C. 1.040 seconds

D. 2.020 seconds

E. 2.080 seconds

F. 10.020 seconds

[✓ **G.** 10.080 seconds]

40 · 10 = 400

✓

6/6

10. [6 points]: Ben notices that his protocol at the receiver is delivering segments to the application at a rate of less than half a segment per second. What's the best way for him to increase that rate?

(Circle the BEST answer)

A. Increase the sender's window size from 10 to 20 segments.

B. Decrease the sender's timeout interval from 5 to 2 seconds.

[✓ **C.**] Increase the sender's timeout interval from 5 to 50 seconds.

D. Increase the router's maximum queue length from 40 to 80 packets.

E. None of the above will help.

✓

6/6

III Atomicity

Ben Bitdiddle is building a transactional file system that can make updates to several files appear to be a single, atomic action. He decides to implement his system using a mechanism similar to shadow copies of files we discussed in class, which he calls *shadow directories*. Similar to a shadow copy of a file, a shadow directory works by having the file system create a copy of a directory and all of its contents before changing any of the files in that directory, and then using an atomic rename operation to install the new directory at commit time.

Ben's implementation is layered on top of the ordinary Unix file system calls. You may assume that the Unix file system provides atomic implementations of link, unlink, and rename, as well as atomic reads and writes of single disk sectors.

Ben begins by trying to build a system that provides all-or-nothing atomicity (i.e., if the system crashes either all changes happen or none of them do) without isolation (i.e., where only one transaction runs at a time.) Ben's initial implementation is shown on the next page, where each function is named `Txxx` to indicate that it is a transactional implementation. The `Trecover` procedure is run after the system crashes and restarts, and before any other file system commands are processed. For brevity, we use the commands `doWrite` and `doRead`; these open the specified file, seek to the specify offset, write or read the specified bytes, and close the file. Assume that there is also a way for transactions to create and delete files, which we do not show. Also assume that directories contain only files (not subdirectories).

```
    // Assume that the character "_" is never used in a user-supplied file or directory name
    // ++ concatenates strings and converts ints to strings

    function Tbegin(directory):
a.      mkdir("new_" ++ directory)
        for each file in directory:
             copy (directory ++ "/" ++ file, "new_" ++ directory ++ "/" ++ file)

    function Tcommit(directory):
b.      rename(directory,"junkdir")
c.      rename("new_" ++ directory, directory)
        delete "junkdir" and its contents

    function Trecover(directory):
      if (not exists(directory))
d.        rename("new_" ++ directory, directory)
      if (exists("new_" ++ directory))
          delete "new_" ++ directory and its contents
      if (exists ("junkdir"))
          delete "junkdir" and its contents

    function Twrite(directory, file, bytes, offset, len):
       fpath = "new_" ++ directory ++ "/" ++ file
       doWrite(fpath,bytes,offset,len)

    function Tread(directory, file, bytes,offset,len):
      fpath = "new_" ++ directory ++ "/" ++ file
      doRead(fpath,bytes,offset,len)
```

11. [6 points]:

[✓ **True**] / **False** Ben's implementation ensures all-or-nothing atomicity in the face of system crashes, assuming there is only one transaction running at a time. ✓

6/6

12. [6 points]: After which line in the above code is the commit point of Ben's implementation? **(Circle the BEST answer)**

A. Line a.

[✓ **B.**] Line b.

C. Line c.

D. Line d.

✓

6/6

So far, Ben has assumed there is only one transaction running at a time.

Ben asks his friend Dana Bass to help him add support for concurrent transactions to his implementation. Dana proposes that Ben modify his code so that it creates a temporary directory `tmp_directory_TID` to contain the intermediate (non-committed) state of each transaction while it runs (where `TID` is a unique identifier assigned to each transaction before it begins); this will prevent concurrent transactions from seeing other concurrent transaction's uncommitted updates.

Dana also proposes keeping multiple versions of the directory around. The idea is that the system will increment a *version number* after each transaction runs, and that each new version will reflect the changes made by one transaction. Successive transactions will start from the files representing the most recent committed version before they began. She allocates a special *version sector* on disk, which contains the current version number. Because it is only one sector, the version sector can be read and written atomically.

She suggests the following implementation (the implementation of `Trecover` is omitted for brevity; we are not asking you to analyze the behavior of this code in the face of crashes):

```
// T is a data structure containing info about current transaction, created by Tbegin

function Tbegin(TID, directory):
  T.TID = TID
  T.dir = directory
  T.vers = read version sector
  // name of a directory for the version this transaction is reading
  T.versDir = T.dir ++ "_" ++ T.vers ++ "/"
  // name of a temporary directory used by a transaction
  T.tmpDir = "tmp_"  ++ T.dir ++ "_" ++ T.TID ++ "/"
  T.changed = {}

  mkdir(T.tmpDir)
  for each file in T.versDir:
     copy(T.versDir++file, T.tmpDir++file)
  return T

function Twrite(T, file, bytes, offset, len):
 doWrite(T.tmpDir++file,bytes,offset,len)
 T.changed = T.changed ∪ file

function Tread(T, file, bytes,offset,len):
  // read from temporary directory so transaction sees its own updates
 doRead(T.tmpDir++file,bytes,offset,len)

function Tcommit(T):
  acquire(commitLock) // only one committer at a time
  vers = read version sector

  if (vers != T.vers):   // get changes from transactions that committed while we ran
    latestVersDir = T.dir ++ "_" ++ vers ++ "/"
    for each file in latestVersDir:
      if (file not in T.changed)
        copy(T.latestVersDir++file, T.tmpDir++file)

  newVers = vers+1
  newVersDir = T.dir ++ "_" ++ newVers ++ "/"
  rename(T.tmpDir, newVersDir)
  write newVers into version sector
  release(commitLock)
```

Note that transactions in Dana's implementation do not follow two phase locking (in fact, no locks are acquired at all in `Tread` *or* `Twrite`!)

Unfortunately, Ben runs this version of the code and finds that it doesn't ensure serializable execution.

13. [10 points]: Which of the following statements about Dana's code are true (assume that if two transactions both write to the same file, they write different data to that file, and that a write may depend on any data read prior to that write.)

(Circle True or False for each choice.)

A. [✓ **True**] / **False** If Dana's code were modified to use the a two-phase locking protocol where it acquires a lock on a file (covering all versions of that file) in `Twrite` or `Tread` before reading/writing the file and releases locks only after commit, it would be serializable.

X

B. **True** / [✓ **False**] Dana's code does not ensure serializability because one transaction may see another transaction's writes before that other transaction has committed.

✓

C. [✓ **True**] / **False** If Dana's code were modified to abort during the execution of `Tcommit` when `vers != T.vers`, her code would be serializable.

✓

D. [✓ **True**] / **False** Dana's code does not ensure serializability because if two transactions both read and update the same file, both of them may read a version of the file that does not include either of their changes.

✓

7/10

Sometimes Ben notices that Dana's implementation *does* result in a serial equivalent ordering of transactions. For each of the following transaction interleavings generated by Dana's code indicate whether it represents a serial-equivalent execution, and if so, indicate the equivalent ordering. Assume that these transaction schedules run to completion and there are no crashes or aborts. Here "R f1" or "W f1" indicates a transaction executed `Tread(T,f1...)` or `Twrite(T,f1,...)`; assume that if two transactions both write to the same file, they write different data to that file, and that a write may depend on any data read prior to that write.

A.

T1	T2
Tbegin(T1,d)	
	Tbegin(T2,d)
R f1	
W f1	
	R f1
	W f1
Tcommit(T)	
	Tcommit(T)

B.

T1	T2
Tbegin(T1,d)	
	Tbegin(T2,d)
R f1	
	R f2
W f2	
	W f1
Tcommit(T)	
	Tcommit(T)

C.

T1	T2
Tbegin(T1,d)	
	Tbegin(T2,d)
R f2	
W f2	
	R f1
W f1	
Tcommit(T)	
	W f3
	Tcommit(T)

14. [8 points]: Write a serial equivalent schedule for each interleaving, or circle "Not Serializable".

A) T1, T2 ×

B) Not serializable ✓

3/8

C) Not serializable ×

End of Quiz II

Please ensure that you wrote your name on the front of the quiz,
and circled your recitation section number.