Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2009

Quiz II

There are 14 questions and 13 pages in this quiz booklet. Answer each question according to the instructions given. You have **50 minutes** to answer the questions.

Most questions are multiple-choice questions. Next to each choice, circle the word **True** or **False**, as appropriate. A correct choice will earn positive points, and a wrong choice or no choice will score 0. Some questions are harder than others and some questions earn more points than others—you may want to skim all questions before starting.

If you find a question ambiguous, be sure to write down any assumptions you make. **Be neat and legible.** If we can't understand your answer, we can't give you credit!

Write your name in the space below AND at the bottom of each page of this booklet.

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.
NO PHONES, NO COMPUTERS, NO LAPTOPS, NO PDAS, ETC.

CIRCLE your recitation section number:

10:00	1. Girod/Badirkhanli	
11:00	2. Girod/Badirkhanli	3. Zeldovich/Reid
12:00		4. Zeldovich/Reid
1:00	5. Jackson/Benjamin	6. Newton/Dowgun
2:00	7. Jackson/Benjamin	7. Newton/Dowgun

Do not write in the boxes below

1-4 (xx/32)	5-6 (xx/14)	7-10 (xx/24)	11-14 (xx/30)	Total (xx/100)

Name:

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.)

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.

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.

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

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.

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.

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.

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.

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.

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.

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.

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.)

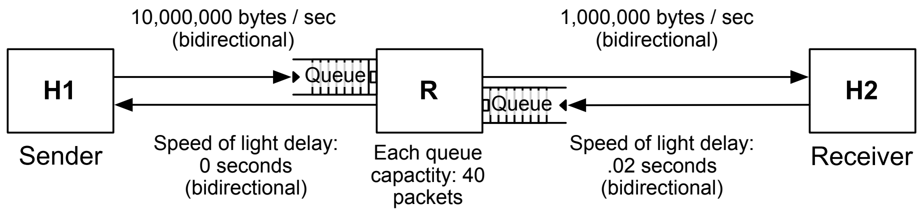
A. True / False Latency of `read()` system calls on the client may be lower.

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

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:


10,000,000 bytes / sec
(bidirectional)
1,000,000 bytes / sec
(bidirectional)
H1
Queue
R
Queue
H2
Sender
Speed of light delay: 0 seconds (bidirectional)
Each queue capactity: 40 packets
Speed of light delay: .02 seconds (bidirectional)
Receiver

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.

Name:

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

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.

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

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.

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.

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.

Name:

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`!)

Name:

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.

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.

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. Schedule:__________________________ Not Serializable

B. Schedule:__________________________ Not Serializable

C. Schedule:__________________________ 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.

Name: