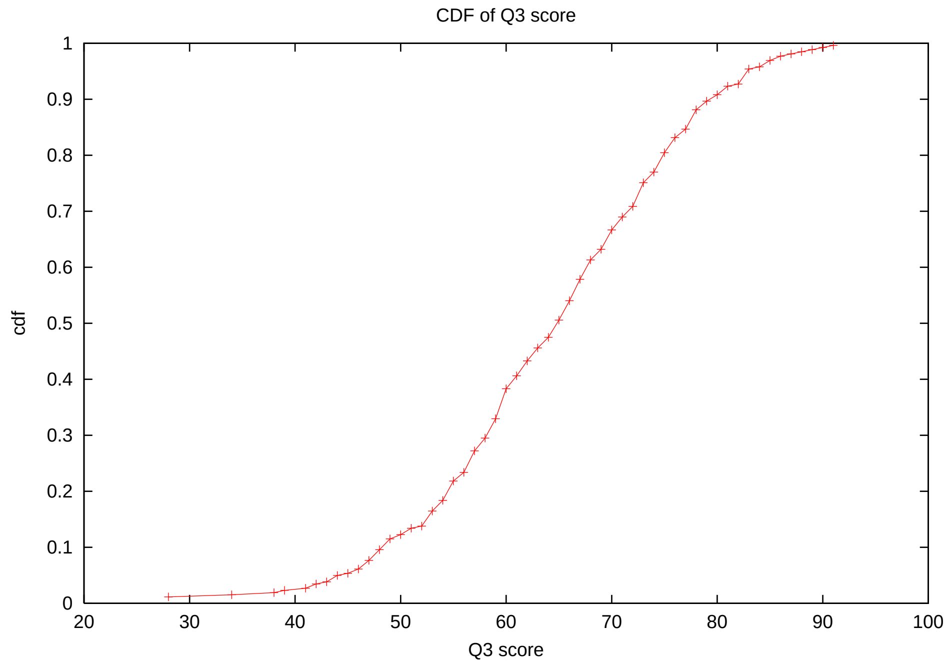
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2003

Handout 31 - Quiz III

All problems on this quiz are multiple-choice questions. In order to receive credit you must fill in the blank(s) or mark the correct answer or answers for each question. You have 50 minutes to answer this quiz.



Name: Solutions

I Reading questions

1. [10 points]: In system R (Reading #19), the LOG file is

(Circle ALL that apply)

A. shadowed
NO. The log is not a file of the file system, it is managed with special software in a separate disk partition. In system R, shadowing is available only to files managed by the file system.

B. mirrored
YES or NO. The log is mirrored, which is the same as duplexed and this feature is optional in System R.

C. duplexed
YES or NO. The log is duplexed, which is the same as mirrored.

D. archived
YES or NO. The log is not regularly archived, but under certain overflow conditions, parts of it are archived..

E. forced at every COMMIT
YES. The log is forced at every commit to preserve the commit across failures.

F. marked as a "winner" during a crash recovery
NO. Transactions, not logs, are marked as winners and losers during crash recovery.

2. [10 points]: Consider each of the following benchmark workloads. For each workload, circle LFS if the LFS file system will complete the test first. Circle FFS if the Fast File System will complete the test first. Circle Tie if neither of the two file systems has a clear performance advantage.

Assume that FFS and LFS work as described in Reading #16, but with a segment size of 100 megabyte. The machine performing the benchmarks has a 100-megabyte buffer cache for data blocks and enough RAM to cache the LFS inode table. Each test starts with an empty file system and you can ignore cleaning of segments.

Some tests require a preparation phase. For your answers, however, you should consider only how the file systems perform on the test phase.

(Circle the right answers)

A. Preparation: create a 1-gigabyte file sequentially. Test: read the file sequentially.

LFS **FFS** **Tie**

TIE. Both FFS and LFS are designed to allocate sequentially written files in large sequential chunks on the disk. This maximizes the performance of sequential reads of the file, and there is, to a first approximation, no reason to prefer one over the other for this task.

B. Preparation: create a 1-gigabyte file; re-write all 1-kilobyte blocks of the file in a pseudo-random order. Test: Read the modified 1-gigabyte file sequentially.

LFS **FFS** **Tie**

FFS. When the file is rewritten, LFS will record the new values of the disk blocks at the end of *the log in the order in which they were written, which is not the sequential order. FFS will simply overwrite the values in the old disk blocks. When the sequential read happens, FFS will read the blocks in the same order that they were originally written on the disk, minimizing seeks. LFS will be forced to jump around the blocks appended to the log in some random order. FFS should perform better.*

C. Preparation: Seed the pseudo-random number generator with "6033"; create a 1-gigabyte file; re-write all 1-kilobyte blocks of the file in a pseudo-random order. Test: Seed the pseudo-random number generator with "6033"; read 1-kilobyte blocks from the modified 1-gigabyte file in the same pseudo-random order.

LFS **FFS** **Tie**

LFS. The read will read the blocks in the same order as they were rewritten. In LFS, the blocks will have been written in this order at the end of the log and the reads will happen in the same order as the blocks appear on the disk, maximizing read performance. In FFS, the blocks will be overwritten, and the read will jump around the sequentially ordered blocks on disk. LFS should perform better.

D. Preparation: Write a 1-gigabyte file sequentially. Test: Read 100,000 1-kilobyte blocks from the file at random locations.

LFS **FFS** **Tie**

TIE. Both file systems will jump around the sequentially ordered disk blocks at random. Both will have the same bad performance.

3. [5 points]: The high performance implementation of a transaction system described in lecture uses undo records to?

(Circle BEST answer)

A. decouple commit from recovery
NO. The job of recovery is to make sure that the effects of committed transactions persist across failures. Commit and recovery are never coupled together in the first place.

B. decouple cache replacement from commit
YES. Undos make it possible to write back a cached page to stable storage before the transactions modifying it have committed. If there is a crash and the transactions abort, during recovery the undos will undo the changes that the aborted transactions made to the copies written back to stable storage.

C. decouple redo of actions from recovery
NO. Even with undos, the recovery mechanism must still execute redos to ensure that the effects of any committed transactions on blocks that were not written out to stable storage are correctly reflected in the recovered state.

D. decouple recovery from install.
NO. The recovery must still apply the proper sequence of undos and redos to restore a correct state from the potentially incorrect state that results from the installs the database system executed.

4. [5 points]: In lecture, Professor Jackson and Saltzer agree with Brooks's "mythical man-month" on

(Circle ALL that apply)

A. conceptual integrity is desirable in a design
YES.

B. it is difficult to come up with the right abstractions for a system
YES.

C. information flow diagrams are an essential design component
NO. We did not discuss information flow diagrams in 6.033. Prof. Jackson talked about dependence diagrams and object models (static structure diagrams).

D. in general a designer's second system comes out better than the first
NO. The general wisdom is that the second system is usually WORSE than the first system.

II Feature Interaction

Alice decided to try out MIT's new RFID Student Tracking System, so she created an access control list that allows a few close friends to track her. One of those friends, Bob, wants to ask Alice to join his 6.033 design project team, so this morning he requested that the tracking system give him a callback if Alice walks by the Great Sail. Alice, working in the Athena Cluster in 66-080, belatedly realizes that Bob is probably going to pop that question, so she logs in to the tracking system and removes Bob from her access control list. She then logs out and leaves for lunch. As she walks by the Great Sail, Bob comes running out of the library to greet her, saying that he just received a callback from the tracking system.

5. [10 points]: The designer of the tracking system made a security blunder. What is it?
(Circle the one most likely answer)

A. The tracking system didn't properly erase residues.
NO. It describes weaknesses that would require an attack to exploit, but the question indicates that Bob received the call-back in the ordinary course of business, so we can exclude this answer.

B. In her rush to leave for lunch, Alice removed Lucy, rather than Bob, from her ACL.
NO. The question specifically asks about a blunder made by the system designer, but B is a blunder made by Alice.

C. The tracking system has a time-of-check to time-of-use bug.
YES. The system checked for permission when Bob issued his query, but did not check when actually performing the action he requested. In between these two times, the access control list changed, resulting in a classic case of the time-of-check to time-of-use bug.

D. The system used a version of SSL that is subject to cipher substitution attacks.
NO. See answer for A.

E. The system did not require a face-to-face rendezvous between users and system administrator.
NO. See answer for A.

III Reconciling file system replicas

After reading the CODA paper and the end-to-end paper, Alice explores designing an application for reconciling UNIX file systems. Her program, *reconcile*, takes as input the names of two directory trees and attempts to reconcile the differences between the two trees. The typical scenario for *reconcile* is that one of the directory trees is on a server (e.g., an Athena locker). The other one is a replica of the server's directory tree, located on Alice's laptop.

When Alice is disconnected from MIT, she modifies files on her laptop while her friends may modify files on the Athena server. When Alice reconnects to MIT, she runs *reconcile* to reconcile the differences between the directory tree on her laptop and the server so that they are identical again. For example, if a file has changed on her laptop, but not on the server, *reconcile* copies the file from the laptop to the server. If the file has changed on both the laptop and server, then *reconcile* requires guidance to resolve conflicting changes.

The *reconcile* program maintains on each host a database (*fsinfo*), which is stored outside the directory tree being reconciled. This database is indexed by pathname, and it stores:

```
char  pathname[1024]; // pathname for this file
int160 hash; // hash = SHA1(content of file's data)
```

On disk a UNIX file consists of metadata (the inode) and content (the data blocks). The SHA1 is computed over the file's content. Pathnames are less than 1024 bytes. For this problem, you can ignore the details of reconciling directories, and assume that Alice has permission to read and write everything in both directory trees.

The *reconcile* program operates in 4 phases:

- Phase 1: Compute the set of changes on the laptop since the last reconciliation and the set of changes on the server since the last reconciliation.
- Phase 2: The laptop retrieves the set of changes from the server. Using the two change sets, the laptop computes a set of actions that must be taken to reconcile the two directory trees. In this phase, *reconcile* might decide that some files cannot be reconciled, because of conflicting changes.
- Phase 3: The laptop carries out the actions determined in phase 2. The laptop updates the files and directories in its local directory tree, and retrieves files, sends files, and sends instructions to the server to update its directory tree.
- Phase 4: Both the laptop and the server update the *fsinfo* they used to reflect the new content of files that were successfully reconciled on this run.

For the rest of the quiz you may assume that *reconcile* runs to completion before starting again. Furthermore, you may assume that when *reconcile* runs no other processes are allowed to modify the file system. You also may assume that initially the *fsinfo* databases are identical in content and computed correctly, and that after reconciliation they also end up in an identical state.

Name:

The first phase of *reconcile* runs the procedure *computeModifications* on both the laptop and the server. On each machine *computeModifications* produces two sets: a set of files that changed on that machine and a set of files that were deleted on that machine.

$set\ changeset, deleteset;$

procedure COMPUTEMODIFICATIONS $(path, fsinfo)$
{
  $changeset \leftarrow NULL;$
  $deleteset \leftarrow NULL;$
  $computeChangeSet(path, fsinfo);$
  $computeDeleteSet(fsinfo);$
}

procedure COMPUTECHANGESET $(path, fsinfo)$
{
  $info \leftarrow lookup(path, fsinfo)$
  if $info == NULL$ **then** $add(path, changeset)$
  else if $isDirectory(path)$ **then** {
    for each $file \in path$ **do** $computeChangeSet(path/file, fsinfo)$
  }
  else if $SHA1(content(path)) \neq info.hash$ **then** $add(path, changeset)$
}

procedure COMPUTEDELETESET $(fsinfo)$
{
  for each $entry \in fsinfo$ **do** {
    if $not(exist(entry.pathname))$ **then** $add(pathname, deleteset)$
  }
}

The *computeModifications* procedure takes as arguments the pathname of the directory tree to be reconciled and the *fsinfo* database. The procedure *computeChangeSet* walks the directory tree and checks if a file was created or changed since the last time *reconcile* ran. SHA1 is an example of a cryptographic hash function, which has the property that it is computationally infeasible to find two different inputs i and j such that $SHA1(i) = SHA1(j)$

The *computeDeleteSet* procedure checks for each entry in the database if the corresponding file still exists; if not, it adds it to the set of files that have been deleted since the last run of *reconcile*.

6. [10 points]: What files will *reconcile* add to $changeset$ or $deleteset$?
(Circle ALL that apply)

A. Files whose content has decayed.
YES. A file whose content has decayed (some bits changed due to, for example, gamma radiation) will pass "SHA1(content(path)) != info.hash", and thus be added to $changeset$.

B. Files whose content has been modified.
YES. Same reason as above.

C. Files that have been created.
YES. A file that has been created since the last reconciliation will pass the "info == NULL" test and be added to the $changeset$.

D. Files whose inode have been modified.
NO. A file whose inode has been modified will not pass the SHA1(content(path) != info.hash" test, because the SHA-1 is computed over the content of the file, not the attributes. This file won't be added to $changeset$*. Since the file exists, it won't be added to the* $deleteset$.

E. Files that have been deleted.
YES. A file that has been deleted will be detected by the computeDeleteSet procedure, and added to the $deleteset$.

F. Files that have been deleted but recreated with identical content.
NO. A file that has been deleted but recreated (with the same pathname) with identical content will not be added to the $changeset$ *because the SHA1(content(path) != info.hash" test won't succeed. the computeDeleteSet procedure will not add the file to the* $deleteset$*, the test won't succeed.*

G. Files that have been renamed.
YES. Files that have been renamed will pass the "info == NULL" test and processed as newly created files, and will be added to $changeset$.

The second phase of *reconcile* compares the two *changeset*s and the two *deleteset*s to compute a set of actions for reconciling the two directory trees. To avoid confusion we call the *changeset* on the laptop, *changeLeft*, and *changeset* on the server, *changeRight*. Similarly, the *deleteset* on the laptop is *deleteLeft* and *deleteset* on the server *deleteRight*. The second phase consists of running the procedure *computeActions* with the 4 sets as input; it produces 5 sets:

- *additionsLeft*: files that must be copied from server to the laptop
- *additionsRight*: files that must be copied from laptop to the server
- *removeLeft*: file that must be removed from laptop
- *removeRight*: file that must be removed from server
- *conflicts*: files that have conflicting changes

In the following code fragment, the notation $A - B$ indicates all elements of the set A after removing the elements that also occur in the set B. With this notation, the 5 sets are computed as follows:

Name:

```
conflicts ← NULL;
procedure COMPUTEACTIONS(changeLeft, changeRight, deleteLeft, deleteRight)
 {
     for each file ∈ (changeLeft)  do {
         if file ∈ (changeRight ∪ deleteRight)  then add(file, conflicts)
      }
     for each file ∈ (deleteLeft)  do {
         if file ∈ (changeRight)  then add(file, conflicts)
      }
     additionsRight ← changeLeft − conflicts;
     additionsLeft ← changeRight − conflicts;
     removeRight ← deleteLeft − conflicts;
     removeLeft ← deleteRight − conflicts;
  }
```

7. [10 points]: What files end up in the set *additionsRight*?

(Circle ALL that apply)

A. Files created on the laptop that don't exist on the server
YES. Files created on the laptop will be added to changeLeft, and won't appear in changeRight or deleteRight, because the files don't exist on the server. Thus, these files will be added to additionsRight.

B. Files that have been removed on the server but not changed on the laptop
NO. Files removed might be added to a remove set, but not to an additions set.

C. Files that have been removed on the laptop but not on the sever.
NO. Same reasoning as 7.B

D. Files that have been modified on the laptop but not on the server
YES. These files will be in changeLeft and not in changeRight or deleteRight, and thus added to additionsRight.

E. Files that have been modified on the laptop and on the server
NO. These files will end up in the conflict list since these files show up in both changeLeft and changeRight.

8. [10 points]: What files end up in the set *conflicts*?

(Circle ALL that apply)

A. Files that have been modified on the laptop and on the server
YES. see 7.E

B. Files that have been removed on the laptop but that exist unmodified on the server
NO. These files are in the changeLeft set, but not in the changeRight or deleteRight, thus they won't be added to conflicts.

C. Files that have been removed on the laptop and on the server.
NO. These files are in the deleteLeft and deleteRight, and the second "foreach" loop will not add them to conflicts.

D. Files that have been modified on the server but not on the laptop
NO. These files are only in the changeRight set, and thus won't be added to conflicts.

E. Files that have been created on the laptop and on the server but with different content
YES. These files will be in the changeRight and changeLeft sets, and thus in the conflicts set.

F. Files that have been created on the laptop and on the server with the same content.
YES. These files, if they have the same pathnames, will be in the changeRight and changeLeft sets, and thus in the conflicts set.

Phase 3 of the *reconcile* executes the actions: deleting files, transfering files, and resolving conflicts. From now on, you can assume that conflicts are resolved by asking the user.

We focus on transfering files from laptop to server. Alice wants to ensure that transfers of files are atomic. You may assume that all file system calls execute atomically. The *reconcile* program transfers files from *additionsRight* by invoking the remote procedure *receive* on the server:

$boolean$**procedure** RECEIVE$(data, size, path)$
{
  $name \leftarrow uniquename();$
  $fd \leftarrow creat(name);$
  if $fd >= 0$ **then** {
    $n \leftarrow write(fd, data, size);$
    $close(fd);$
    if $n == size$ **then** $rename(name, path);$
    else $unlink(name);$
    $return(n == size);$
  }
  else $return(false);$
}

The *receive* procedure takes as argument the new content of the file (*data* and *size*) and the name of the file to be updated or created. First, *receive* creates a temporary file with some unique name (*name*) and writes the data into in it. After the write is successful, *receive* renames the temporary file to its real name (*path*), removing the old version of *path* first, if it exists; otherwise, it cleans up and deletes the temporary file. (You may assume that *rename* always executes successfully.)

9. [5 points]: Where is the commit point in the procedure *receive*?
(Circle the BEST answer)

A. right after *rename* completes
B. right after *close* completes
C. right after *create* completes
D. right after *unlink* completes
E. right after *write* completes
F. none of the above

Depending on your interpretation of "commit point", the correct answer is either A or F. Receive commits by executing rename; it aborts by executing unlink. Since file system operations are said to be atomic, the commit point is either when rename returns or when unlink returns, so the best answer is F. However, if one considers the term "commit point" to apply only when the operation commits, then A is the best answer.

After the server or laptop fails, it calls a recovery procedure to back out or roll forward a *receive* operation that was in progress when the host failed.

10. [5 points]: What must this recovery procedure do?

(Circle the BEST answer)

A. Remove any temporary files left by *receive*

B. Nothing

C. Send a message to the sender to restart the file transfer

D. Rename any temporary files left by *receive* to their corresponding pathname

A (or with explanation, B). Alice wants RECEIVE to be atomic, which means that if it fails to reach its commit point it should leave no evidence that it was ever there. If a crash occurs after RECEIVE calls creat and before RECEIVE reaches its commit point, there will be a uniquely named temporary file that may persist indefinitely unless the recovery procedure removes it. In some file systems, if you place the temporary file in a special directory intended for temporary files, a system recovery procedure will automatically remove it, which would be a suitable explanation for circling answer B. Answer C is something that could be done, but is not required to achieve atomicity. Answer D is incorrect because rename is the mechanism to commit the transfer. The crash may have happened in the middle of the file transfer, in which case the transfer should not be committed.

11. [10 points]: Which advantages does the *reconcile* program have over the CODA reconciliation procedure (as described in reading #20)

(Circle ALL that apply)

A. *reconcile* repairs files that decay
NO. In contrast with CODA, reconcile does detect decayed files, but unfortunately it replaces the good file with the bad one, which is not the usual meaning of "repair".

B. *reconcile* doesn't require changes to the UNIX file system implementation
YES. Reconcile can run on a standard UNIX file system, while CODA requires a file system that keeps a log of every operation, which the UNIX file system does not do.

C. *reconcile* doesn't require a log on the laptop
YES. CODA reconciliation requires that the laptop keep a running log, while reconcile needs only to remember an unchanging file between one run and the next.

D. *reconcile* propagates changes from the laptop to the server, and vice versa.
NO. It is a true statement, but CODA does the same thing, so it isn't an advantage.

E. *reconcile* will run much faster on big files
NO. It is a false statement. Reconcile calculates a SHA-1 hash of every file, which requires reading everything in the file system, while CODA simply looks in its log to see what, if anything, has changed, so CODA would be expected to do reconciliations much faster, especially on big files that haven't changed.

Alice wonders if her code extends to reconciling more than two file systems. Consider 3 hosts (A, B, and C) that all have an identical copy of a file f, and the following sequence of events:

- at noon B modifies file f
- at 1pm B reconciles with A
- at 2pm C modifies f
- at 3pm B reconciles with C
- at 4pm A modifies f
- at 5pm B reconciles with A

You should assume that B has two *fsinfo* databases, one used for reconciling with A and one for reconciling with C.

12. [10 points]: Which of the following statements are correct, given this sequence of events and Alice's implementation of *reconcile*?

(Circle ALL answers that apply)

A. If the conflict at 3pm is reconciled in favor of B's copy, then *reconcile* will not report a conflict at 5pm.

B. If the conflict at 3pm is reconciled in favor of C's copy, then *reconcile* will report a conflict at 5pm.

C. If the conflict at 3pm is resolved by a modification to f that merges B's and C's versions, then *reconcile* will report a conflict at 5pm.

D. If the conflict at 3pm is resolved by removing f from B and C, then *reconcile* will not report a conflict at 5pm.

Answer is A B and C but not D. To avoid confusion between scenario identifiers and participants, call the participants Alyssa, Ben, and Charlie. The original file is f, and its hash is found in both A-Bfsinfo and B-Cfsinfo. Ben creates state f' at noon, Charlie creates state f'' at 2 p.m, and Alyssa creates state f''' at 4 p.m. At the 1 p.m. Alyssa-Ben reconciliation, Alyssa receives and A-Bfsinfo records state f'. The 3 p.m. Ben-Charlie reconciliation discovers that states f' and f'' are both different from state f as found in B-Cinfo, so there is a conflict whose outcome depends on the scenario:

Scenario A: Charlie receives state f'.
Scenario B: Ben receives state f''.
Scenario C: Ben and Charlie receive state f''''.
Scenario D: Ben and Charlie do not have an f.

In each scenario, B-Cfsinfo records the new state (or absence) of f, but that is irrelevant to the question, because the 5 p.m. reconciliation uses A-Bfsinfo, which still shows file f at state f'.

We have the following results:

	Alyssa	A-Bfsinfo	Ben	report	answer
Scenario A	f'''	f'	f'	no conflict	circle A
Scenario B	f'''	f'	f''	conflict	circle B
Scenario C	f'''	f'	f''''	conflict	circle C
Scenario D	f'''	f'	absent	conflict	don't circle D

End of Quiz III — enjoy your summer!