Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2004

Quiz III Solutions

There are 15 questions and 13 pages in this quiz solutions booklet. To receive credit for a question, answer it according to the instructions given. *You can receive partial credit on questions if you circle some of the correct answers, but we may subtract points for circling a wrong answer.* This quiz is a standard 50-minute quiz, but you have **90 minutes** to answer the questions.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be harder than others. Attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

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

Do not write in the boxes below

1-3 (xx/24)	4-6 (xx/22)	7-11 (xx/22)	12-13 (xx/16)	14-15 (xx/16)	Total (xx/100)

Name: Solutions

I Pot-pourri

1. [8 points]: Which of the following statements is true of the crash recovery procedure in the log-structured file system (LFS) as described by Rosenblum and Ousterhout in reading #14?

(Circle ALL that apply)

(A.) For correct crash recovery, LFS's checkpoint region (which includes the time of the checkpoint) must be written in a recoverable manner.

The recovery procedure starts with information in the checkpoint. If the information contained in the checkpoint region was incorrectly written (e.g., it was partially written because of a non-recoverable write), then the crash recovery procedure would produce the wrong result.

(B.) To be able to successfully recover a file's data blocks during roll-forward, LFS must write a file's blocks before writing the file's inode to the log.

LFS's data structures build on those of Unix. The inode refers to specific blocks on disk. If an error occurred after the inode was written but before the blocks themselves were written, then roll-forward would read the (out of date or uninitialized) blocks referred to by the inode.

C. The roll-forward procedure ensures that all file blocks that were written before a crash will be visible after recovery.

Blocks are written before inodes. If a crash occurs after a block is written but before that inode that refers to it is written, then the block is not visible after recovery.

(D.) Crash recovery time typically decreases if checkpoints are made more frequently.

Crash recovery works forward from the most recently written checkpoint to the end of the log. If checkpoints are made more frequently, then (on average) there is less log after the most recently written checkpoint.

2. [8 points]: Which of the following statements is true of System R (reading #17, *The Recovery Manager of the System R Database Manager*)?

(Circle ALL that apply)

(A.) During recovery from a crash, System R undoes any losers that may have saved data at the last checkpoint, as well as redoing the actions of committed winners that were made since the last checkpoint.

This is the undo/redo recovery scheme described in the System R paper.

B. During recovery from a crash, System R does not need to redo any winners that committed because each transaction COMMIT SAVEs file data on disk.

In System R, COMMIT and SAVE are independent. In System R, file SAVE is called only by the system, and only at checkpoints. So the only thing guaranteed to be on disk for a committed transaction is the log.

C. System R stores its log only in volatile memory for high performance and relies on shadow files for crash recovery.

System R initially writes its log to volatile memory, but at transaction commit time it flushes the log to non-volatile (disk) memory. Shadow pages are not sufficient for crash recovery because

there can be multiple transactions that have made changes to the same file, some of which committed and others of which need to be undone.

D. System R could invoke a checkpoint operation in the middle of a transaction.

When recovering from the checkpoint, some of the transaction's operations may need to be undone or redone (depending on whether the transaction is a winner or a loser). The authors of the System R paper argue that quiescing the system (i.e., finding or forcing a time when no transactions are in progress) in order to write a checkpoint would result in poor performance.

3. [8 points]: Which of the following techniques are used by the Unison file synchronizer (Trevor Jim, Benjamin C. Pierce, and Jérõme Vouillon, *How to build a file synchronizer*, Unpublished draft, February 22, 2002)?

(Circle ALL that apply)

A. An efficient algorithm to transfer files.

Unison uses the same efficient file transfer algorithm as rsync.

B. Cryptographic fingerprints.

If the fingerprints match, then so do the underlying files (with high probability). It is a more efficient use of scarce network bandwidth to compute the fingerprints at both systems, then transfer just one fingerprint across the network for comparison with the other, than to transfer one of the files and compare it to the other one.

C. A trace of file modifications.

Other reconciliation systems such as Coda trace file modifications, but doing so requires low-level changes to the operating system or to all applications in order to intercept all modifications. Unison eschews such low-level strategies and runs after the fact to reconcile files that have already been modified via any mechanism.

D. Locks to serialize concurrent executions of Unison.

The locks prevent simultaneous modification to the databases and files, which could result in data corruption.

4. [8 points]: Ben Bitdiddle visits a Web site *amazing6033.com* and obtains a fresh page signed with a private key. Which of these methods of obtaining the certificate for the server's public key can assure Ben that the private key used for the page's signature indeed belongs to the organization that owns the domain *amazing6033.com*? (You may assume that the certificate is signed by a trusted certificate authority and is valid.)

(Circle ALL that apply)

(A.) Using HTTP Ben downloads the certificate from *http://amazing6033.com*.

(B.) Using HTTP Ben downloads the certificate from the certificate authority.

(C.) Ben finds the certificate by doing a Web search on Google.

(D.) Ben gets the certificate in email from a spammer.

All options are true. The authenticity of a valid certificate is independent of the source from which it was obtained. No spammer (or other unsavory party) could forge the trusted certificate authority's signature.

5. [8 points]: Which of the following statements is true of the Slammer Internet worm as described in the paper *Slammer: An urgent wake-up call* by Jerome H. Saltzer?

(Circle ALL that apply)

(A.) The worm exploited a buffer overrun vulnerability. *The worm took control of the victim's computer by overwriting the victim's stack, causing execution of a return statement to transfer control into malicious code.*

(B.) The worm's spread would have been faster had there been more vulnerable hosts in the Internet. *The more vulnerable hosts there are, the greater the rate of Infection, because the likelihood of finding a vulnerable host in a random address scan is higher.*

C. The worm's spread would have been faster had it used TCP, because TCP would have recovered from any worm packets that got lost due to network congestion. *The worm's spread was faster because it used UDP, which does not have the connection setup overhead of TCP. Using relatively small UDP packets permitted the worm to send more copies of itself in a given time period and thus infect more hosts. Slammer did not care where it sent its single-packet payload; resending to the same host was no better from its point of view than sending a new packet to another randomly selected host.*

D. The worm made network performance slow by causing the software in Internet routers to crash. *There is no evidence that any router's software crashed due to the Slammer worm.*

Name:

6. [6 points]: Michael Ernst's lecture, Jerry Saltzer's lecture, and Fred Brooks's book *The Mythical Man-Month* discussed why software systems fail. Which of the following are reasons given by these experts?

(Circle ALL that apply)

(A.) Software subsystems, unlike battleships, are redesigned by 25 year-olds who are high on jolt cola at 2 in the morning.

The malleability (ease of modification) of software makes it all too easy, and all too tempting, to change. An engineer should remember that there are no simple changes to complex systems; such changes frequently have large negative repercussions.

(B.) In many organizations, good news travels fast up the management chain, but bad news does not.

No one wants to be the bearer of bad news. This tendency is called the "bad-news diode": it lets good news pass through, but stops bad news. As a result, corrective action is not taken when it could do the most good.

(C.) Project managers add people to the development team of a system that is already behind schedule.

Managers frequently make this mistake, but it hurts as much as it helps. Adding more people to a project that is already late tends to make it even later. First, the new people have to be trained, which takes resources from the real work of the project. Second, pairwise communication among n people grows as $O(n^2)$, so in a larger team, each person spends a greater fraction of his or her time communicating with others rather than doing real work. The notion of man-month is said to be mythical because man-months are not fungible (i.e., they are not all equivalent, and cannot be easily substituted for one another).

II "Log"-ical calendaring

Ally Fant is designing a calendar server to store her appointments. A calendar client contacts the server using the following remote procedure calls (RPCs):

> ADD(*timeslot*, *descr*): Adds the appointment description (*descr*) to the calendar at time slot *timeslot*.
> SHOW(*timeslot*): Reads the appointment at time slot *timeslot* from the calendar and displays it to the user. (If there is no appointment, SHOW displays an empty slot.)

You may assume that the RPC between client and server runs over a transport protocol that provides "at-most-once" semantics.

The server runs on a separate computer and it stores appointments in an append-only log on disk. The server implements ADD in response to the corresponding client request by appending an *appointment entry* to the log. Each appointment entry has the following format:

```
structure appt_entry {
    int  id;           // unique id of action that created this entry
    char timeslot[200];// the timeslot for this appointment
    char descr[200];   // description of this appointment
};
```

Ally would like to make the ADD action atomic. She realizes that she can use RECOVERABLE_PUT(data,sector) and RECOVERABLE_GET(data,sector). These functions guarantee that a recoverable sector is either written completely or not at all. They were discussed in a 6.033 lecture and in chapter 8, page 8-28. (You don't need to study their implementation, however, to answer the questions on this quiz.)

Each appointment entry is for one *timeslot*, which specifies the time interval of the appointment (e.g., 1:30pm-3:00pm on May 20, 2004). Each appointment entry is exactly as large as a single recoverable sector (512 bytes). The first recoverable sector on disk, numbered 0, is the *master_sector*, which stores the recoverable sector number for where the next log record will be written. The number stored in *master_sector* is called the end of the log, *end_of_log*, and is initialized to 1.

Ally designs the following procedure:

procedure ADD($timeslot, descr$)
{
    $id \leftarrow$ ACTION_ID(); *// returns a unique identifier*
    $appt \leftarrow$ MAKE_NEW_APPT($id, timeslot, descr$); *// make and fill in an appt_entry*
    if RECOVERABLE_GET($end\_of\_log, master\_sector$) $\neq$ OK **then return**;
    if RECOVERABLE_PUT($appt, end\_of\_log$) $\neq$ OK **then return**; (1)
    $end\_of\_log \leftarrow end\_of\_log + 1$; (2)
    if RECOVERABLE_PUT($end\_of\_log, master\_sector$) $\neq$ OK **then return**; (3)
}

Name:

The procedure ACTION_ID returns a unique action identifier.

The procedure MAKE_NEW_APPT allocates a *structure appt_entry* object, and fills it in, padding it to 512 bytes.

Ally implements SHOW as follows:

1. Use RECOVERABLE_GET to read the master sector to determine the end of the log.
2. Scan the log backwards from the end using RECOVERABLE_GET on each sector, stopping as soon as an entry for the timeslot is found.
   Note: Our intention was that SHOW started its backward scan from the last written recoverable sector, i.e., from $end\_of\_log - 1$**.**

To help understand if her implementation of the calendar system is correct or not, Ally defines the following properties that her calendar server should ensure:

P1: SHOW(*timeslot*) should display the appointment corresponding to the last committed ADD to the timeslot, even if system crashes occur during calls to ADD.
P2: The calendar server must store the appointments corresponding to all committed ADD actions for at least three years.
P3: If multiple ADD and SHOW actions run concurrently, their execution should be serializable and property P1 should hold.
P4: No ADD should be committed, if it has a time slot that overlaps with an existing appointment.

We learned a number of concepts in 6.033: isolation, recoverability, consistency, durability, and transaction.

7. [1 point]: Which of the above concepts does ADD correctly implement?

(Fill in the BEST term for the blank)

Recoverability. The ADD *code is recoverable; if the system crashes or* ADD *returns, the effect is as if* ADD *either finished adding the appointment fully, or not at all.*
An alternate (less precise) answer to this question is "P1", which holds if there are no concurrent ADD *actions.*

8. [3 points]: For each of the properties P2, P3, and P4, state the concept from 6.033 that describes it the best.

(Fill in the BEST term for each blank)

A. P2: *Durability*

B. P3: *Isolation*

C. P4: *Consistency (P4 is an example of a consistency invariant).*

Name:

9. [8 points]: What is the earliest point in the execution of the ADD procedure that ensures that a subsequent SHOW is guaranteed to observe the changes made by the ADD. (You may assume that SHOW does not fail.)

(Circle BEST answer)

A. The successful completion of RECOVERABLE_PUT in the line labeled (1).
Since the code for the SHOW *procedure is not available for examination, some reasoning is needed to establish how it must work, based on its two-line text description. In particular, we must deduce what "end of the log" means when* SHOW *looks in the log. To start, we have already established that* ADD *provides recoverability, which means that its additions are either fully installed or else not visible to* SHOW *at all. The description of the master sector is that it always contains the sector number where the next log record will be written, which in turn means that the last log record that was successfully written will be in the previous sector. For* ADD *to be recoverable,* SHOW *must begin scanning with the last log record that was successfully written. As a result, it will not see a record that is in the process of being written until* ADD *successfully completes the* RECOVERABLE_PUT *in the line labeled (3). (Conversely, if* SHOW *were to mistakenly start by examining the recoverable sector pointed to by the master sector, it would at some times be looking at a recoverable sector that has never been written, at other times at a recoverable sector that is in the process of being written, and at still other times a sector that was written by an* ADD *that decided to abort.)*

B. The successful completion of the line labeled (2).
No. Same reason as A. Line 2 just updates an internal variable.

(**C.**) The successful completion of RECOVERABLE_PUT in the line labeled (3).
Yes. As soon as RECOVERABLE_PUT *in the line labeled (3) completes, a subsequent* SHOW *will be able to see this change.*

D. As soon as ADD returns.
No. The implied return at the end of the procedure happens later than answer C, so C is a better answer. Moreover, if RECOVERABLE_GET *or* RECOVERABLE_PUT *encounters an error,* ADD *may return without making any change visible to* SHOW.

10. [8 points]: Ally sometimes accesses the calendar server concurrently from different client machines. Which of these statements is true of properties P3 and P4? (Assume that no failures occur, but that the server may be processing multiple RPCs concurrently.)

(Circle ALL that apply)

(**A.**) If exactly one ADD and several SHOW actions run concurrently on the server, then property P3 is satisfied even if those actions are for the same timeslot.
The chooser physical sector for the recoverable master sector controls what SHOW *reads; only* ADD *modifies that chooser physical sector, and* ADD *makes that modification by overwriting. Because one cannot read a disk sector at the same time it is being written, reads and writes to a single sector are naturally serialized, so P3 is satisfied even in the absence of locks.*

B. If more than one ADD and exactly one SHOW run concurrently on the server, then property P3 is satisfied as long as the actions are for different timeslots.
If two ADD *operations run concurrently, they may both read the same value for end_of_log, and then both write their appointment in the same log sector, thus violating P3.*

Name:

C. Suppose ADD(*timeslot*,*descr*) calls SHOW(*timeslot*) before the line labeled (3) and immediately returns to its caller if the timeslot already has an appointment. If multiple ADD and SHOW actions run concurrently on the server, then property P4 is satisfied whether or not property P3 holds.

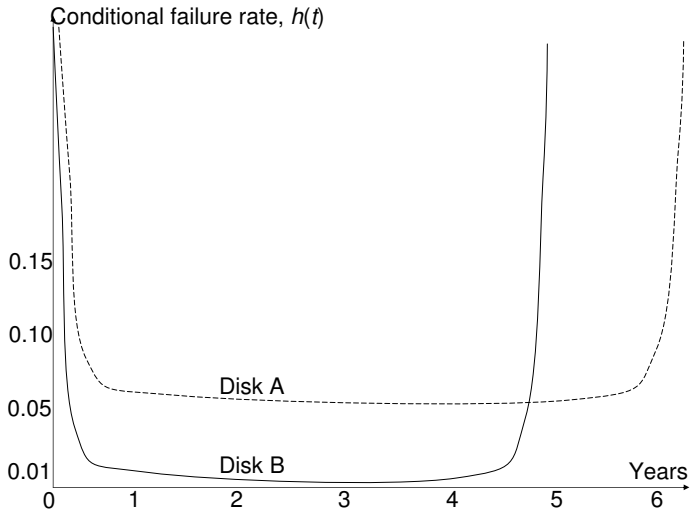
If P3 does not hold, then P4 cannot hold, because two concurrently running ADD *procedures for the same timeslot without proper isolation may end up committing both of them. That violates P4.*

D. Suppose ADD(*timeslot*,*descr*) calls SHOW(*timeslot*) before the line labeled (3) and immediately returns to its caller if the timeslot already has an appointment. If multiple ADD and SHOW actions run concurrently on the server, then property P4 is satisfied as long as property P3 holds.

P3 ensures that a set of concurrent ADD *and* SHOW *actions is equivalent to some serial order. In that serial order, each* ADD *that tries to add an appointment to a timeslot that already has one will not commit. Therefore, P4 will hold if P3 holds.*

11. [2 points]: Ally finds two disks A and B whose conditional failure probabilities follow the "bathtub curve", shown below. Which disk should she buy new to have a higher likelihood of meeting property P2 for at least one year? You may assume that the disk manufacturers sell units that have been "burned in," but otherwise are unused.

(Circle "Disk A" or "Disk B" in the figure below)



Disk B. The disks are said to be burned-in, so they are past the six-month knee of their conditional failure rate curves. The disks are said to be in their first year of service, and the durability requirement is three years, so we care about the probability of failure between 6 months and 4.5 years. Disk B has a lower conditional failure rate over that entire time span.

Name:

Multi-user calendar

Ally becomes president of MIT and opens her calendar on her server to the entire MIT community to add and show entries. People start complaining that it takes a long time for them to SHOW Ally's appointments. Ally's new provost, Lem E. Fixit, tells her that a single log makes reading slow.

Lem convinces Ally to use the log as a recovery log, and use a *volatile in-memory* table, named *table*, to store the appointments to improve the performance of SHOW. The table is indexed by the timeslot. SHOW is now a simple table lookup, keyed by the timeslot.

If the system crashes, the table is lost; when the system recovers, the recovery procedure reinstalls the table. Lem shows Ally how to modify the recovery log to include an "undo" entry in it, as well as a "redo" entry. All the log writes are done using RECOVERABLE_PUT.

Ally writes the following lines in her NEW_ADD pseudocode. (For now, the writes to the log are only shown in COMMIT.)

```
procedure NEW_ADD(timeslot, descr)
 {
     id ← ACTION_ID();                                              (1)
     appt ← MAKE_NEW_APPT(id, timeslot, descr);                     (2)
     table[timeslot] ← appt;                                        (3)
     if (OVERLAPPING(table, appt)) then ABORT(id);                  (4)
     COMMIT(id);                                                    (5)
 }

procedure COMMIT(id)
 {
     if RECOVERABLE_GET(end_of_log, master_sector) ≠ OK then ABORT(id);
     if RECOVERABLE_PUT(["COMMIT", id], end_of_log) ≠ OK then ABORT(id);
     end_of_log ← end_of_log + 1;
     if RECOVERABLE_PUT(end_of_log, master_sector) ≠ OK then ABORT(id);
 }
```

OVERLAPPING checks *table* to see if *appt* overlaps with a previously committed appointment (property P4).

ABORT uses the log to undo any changes to *table* made by NEW_ADD, releases any locks that NEW_ADD set, and then terminates the action.

Ally modifies SHOW to look up an appointment in *table*, instead of scanning the log.

Name:

12. [8 points]: Which of the following statements is true for NEW_ADD with respect to property P1 (P1 is "SHOW should display the appointment corresponding to the last committed ADD.")? (You may assume that there are no concurrent actions.)

(Circle ALL that apply)

(A.) If NEW_ADD writes the log entry corresponding to the *table* write just before the line labeled (3), then P1 holds.

This choice is the write ahead log protocol. If a crash or abort occurs, the log has the information required to undo any changes. Hence, SHOW *will display only the appointment corresponding to the last committed* ADD.

B. If NEW_ADD writes the log entry corresponding to the *table* write just before the line labeled (5), then P1 holds.

The reason this choice does not work is because OVERLAPPING *could abort. If, at the time this abort occurs, the log does not have information to undo the changes made by the* ADD*, then a subsequent* SHOW *may observe the changes made by this uncommitted* ADD*. That violates P1.*

C. Because *table* is in volatile memory, there is no need for ABORT to undo any changes made by NEW_ADD in order for P1 to hold.

See the reason for not circling choice B above.

(D.) If Ally had designed *table* to be in *non-volatile* storage, and NEW_ADD inserts the log entry just before the line labeled (3), then P1 holds.

Write ahead logging is correct whether the cell storage is in volatile or non-volatile memory.

Lem convinces Ally that using locks can be a good way to ensure property P3. Ally uses two locks, ℓ_t and ℓ_g. ℓ_t protects *table*[*timeslot*] and ℓ_g protects accesses to the log. She needs help to figure out where to place the lock acquire and release statements to ensure that property P3 holds when multiple concurrent NEW_ADD and SHOW actions run.

13. [8 points]: Which of the following placements of ACQUIRE and RELEASE statements in NEW_ADD correctly ensures property P3?

(Circle ALL that apply)

This question required the unstated assumption that SHOW *implemented correct locking.*

(A.) ACQUIRE(ℓ_t) just before the line labeled (2),
RELEASE(ℓ_t) just after the line labeled (5),
ACQUIRE(ℓ_g) just before the line labeled (2),
RELEASE(ℓ_g) just after the line labeled (5).

This scheme is simple locking.

B. ACQUIRE(ℓ_t) just before the line labeled (3),
RELEASE(ℓ_t) just after the line labeled (4),
ACQUIRE(ℓ_g) just before the line labeled (5) but after RELEASE(ℓ_t),
RELEASE(ℓ_g) just after the line labeled (5).

This scheme violates two-phase locking, so that gives a hint that something is amiss. If RELEASE(ℓ_t) *happens just after the line labeled (4), then another atomic action may be able to view the*

changes made by this ADD *to the table. Now, if the* ADD *were to* ABORT *inside* COMMIT*, then property P3 is violated.*

C. None of the above.

Disconnected calendar

Ally Fant wants to use her calendar in disconnected operation, for example, from her PDA, cell phone, and laptop. Ally modifies the client software as follows. Just before a client disconnects, the client copies the log from the calendar server atomically, and then reinstalls *table* locally. When the user (i.e., Ally) adds an item, the client runs NEW_ADD on the client, updating the local copy of the log and *table*.

When the client can connect to the calendar server or any other client, it reconciles. When reconciling, one of the two machines is the primary. If a client connects to the calendar server, the server is the primary; if a client connects to another client, then one of them is the primary. The client that is not the primary calls RECONCILE, which runs locally on the client:

```
procedure RECONCILE(primary, client_log)
 {
    for each entry ∈ client_log  do                                          (1)
    {
        if (entry belongs to a COMMITted action)
          then primary.NEW_ADD(entry.timeslot, entry.descr);    // invoke NEW_ADD on primary
     }
    client_log ← primary.COPY_LOG();    // copy log from primary               (2)
    discard local table;
    reinstall table from client_log;
  }
```

You may assume that RECONCILE is atomic and that no crashes occur during reconciliation. Assume also that between any pair of nodes there is at most one active RECONCILE at any time.

14. [8 points]: Which of the following statements is true about the implementation that supports disconnected operation?

(Circle ALL that apply)

(A.) RECONCILE will resolve overlapping appointments in favor of appointments already present on the primary.

NEW_ADD*() will not add any entries to the primary if the entry to be added overlaps with one already present in the primary for that timeslot.*

(B.) Some appointments added on a disconnected client may not appear in the output of SHOW after the reconciliation is completed.

If an appointment added on a disconnected client overlaps with one present on the primary when reconcilation begins, after reconciliation that appointment will not appear in the log or table on either replica.

Name:

C. The result of client C_1 reconciling with client C_2 (with C_2 as the primary), and then reconciling C_2 with the calendar server, is the same as reconciling C_2 with client C_1 (with C_1 as the primary), and then reconciling C_1 with the calendar server.
When C2 is the primary, any overlapping appointments will be reconciled in favor of C2, but when C1 is the primary, those same overlapping appointments will be reconciled in favor of C1. As a result, the server can end up with a different set of appointments in the two cases.

D. Suppose Ally stops making changes, and then reconciles all clients with the server once. After doing that, the logs on all machines will be the same.
All appointments will be present on the primary after all clients reconcile, but clients that reconciled early will not know about appointments that were found on clients that reconciled later.

Lem E. Fixit notices that the procedure RECONCILE is slow. To speed it up, she invents a new kind of record, called the "RECONCILE" record. Each time RECONCILE runs, it appends a RECONCILE record listing the client's unique identifier to the primary's log just before the line labeled (2).

15. [8 points]: Which of the following uses of the RECONCILE record speeds up RECONCILE correctly? (Assume that clients reconcile only with the calendar server.)

(Circle ALL that apply)

(A.) Modify the line labeled (1) to scan the client log backwards (from the end of the log), terminating the scan if a RECONCILE record with the client's identifier is found, and then scan forward until the end of the log calling NEW_ADD on the appointment entries in the log.
NEW_ADD*() will not add any entries to the primary which overlap, so appointments already present on the primary remain after reconcilation.*

B. Modify the line labeled (1) to scan the client log forwards (from the beginning of the log) calling NEW_ADD on the appointment entries in the log, but terminating the scan if a RECONCILE record with the client's identifier is found.
This procedure will miss records added since the last reconciliation.

(C.) Don't reinstall *table* from scratch at the end of reconciliation, but instead update it by adding the entries in the primary log (which the client just copied) that are between the previous RECONCILE record and the RECONCILE record from the current reconciliation. If an entry in the log overlaps with an entry in the table, then replace the table entry with the one in the log.
The only changes that need to be made to the table are new entries contained only in the primary log, or appointments in the client that are removed as a result of reconciliation. Since any conflicting appointment <u>also</u> has a new entry on the primary, no client table entries need to be deleted. All such appointments needing consideration follow the last RECONCILE record. Note that this choice makes the assumption that the "previous RECONCILE record" mentioned in the choice has the same client identifier as the one from the current reconciliation.

D. Assign Lem E. Fixit a different job. None of these optimizations maintains correctness.
Two above optimizations are correct.

End of Quiz III — Enjoy summer!

Name: