DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033—Computer System Engineering Handout #36—May 3, 1999

Practice Quiz #3—Solutions.

This document contains all of the solutions to the old Quiz 3's from 1994 to 1998. The solutions are not in order, to motivate you to solve the problems before looking over the solutions. Some of the provided answers are more verbose than were required, to cover all possible right answers. These are usually noted.

The wide range of question styles is not completely accurate. True/False and multiple choice questions are very likely.

Good luck on the quiz Wednesday. If you have been missing everything, here are the details given on the quiz announcement:

1. Quiz 3 will be from 2-3pm on Wednesday, May 12, 1999.

2. The quiz will be held on the third floor of Walker for students with last names from A-R, and in 34-101 for those with names S-Z.

3. The quiz will cover all the material up to and including the May 10 lecture.

4. There WILL be a quiz review, due to popular demand. It will be held Monday, May 10, from 7-9, in 34-101.

5. Yes, the quiz is open-book, open-notes.

6. **One question from the practice quizzes will be on the quiz.**

*.

Is faster than a RAID level-4 system for workloads that are mostly small writes.

*.

```
                    Seek time:  6 ms.
1/2 revolution rotation time:  4 ms.    7500/60 = 125 rps = 8 ms/revolution
                    Read time:  2 ms.    (4 Kilobytes)/(2 Megabytes/sec.)
                   Total time: 12 ms.
```

*.

As changes are made over time, the log will grow larger and larger. This leads to several disadvantages:

1. Slow recovery, since the entire log must be replayed.
2. Slow reads, since the whole log may need to be examined to find the current value for a variable.
3. A lot of disk space is consumed.

*.

The size of the cache is irrelevant and the access time to the first byte is negligible. Assuming that the system first reads a missing disk block into the cache, and then copies it into the user's read area,

$$T_{av} = [(1-h)*12 + 0.2] \text{ milliseconds}$$

If the system somehow reads the disk block in parallel into both the cache and the user's read area, the second term should be multiplied by h.

*.

```
integer procedure Register-two(H1, H2)
        Seize(H1)
        Seize(H2)
        status = Register(H1)
        if (status== 0) then
                status = Register(H2)
                if (status == -1)
                        Drop(H1)
                endif
        endif
        Release(H1)
        Release(H2)
        return status
```

*.

```
commit(tid) {
append COMMIT record to log
apply updates in intentions list for tid
to database disk and clear list
append COMPLETED record to log
}
```

*.

In order to increase the size of the block cache, Ben must have reduced the memory available for other things such as virtual memory, heap space, or application storage. These things are now working harder to utilize their smaller allocation. If the cache is really big, the time spent searching it may also now be significant.

*.

Overwrite semantics may lead to lost data and prevent recovery!
1. If the disk crashes during a write, a partial write results; the data will be corrupt and unrecoverable.
2. If a commit requires multiple writes and a crash occurs between these writes, there is not enough information to abort the transaction on recovery by undoing its changes. (Similarly, there is not enough information on the disk to handle recovery after a nested transaction commits, when the top-level transaction has not committed: we should abort the nested transaction, but how will we do this?)

*.

This is fundamentally a problem about incommensurate scaling...
1. Don't visit the lab when Louis turns it on for the first time, because it will probably explode. The force required to keep the edges from flying away probably will exceed the tensile strength of the disk platter.
2. Assuming Louis switches to titanium platters to avoid the first problem, the edge of the disk is moving at Mach 1.6, while the innermost track is moving at about Mach 0.5. The seek arm may have some trouble moving across the region of the platter where it crosses the sound barrier.
3. The data will be passing under the read heads an order of magnitude faster than it used to, and they probably will require a complete redesign for the higher bandwidth.
Note also that since the change didn't do anything to improve the seek time, the overall performance may not improve nearly as much as Louis hopes.

*.

new-value from the last logged *WRITE(tid,v,old-value,new-value)* followed by a *COMMIT(tid)*, or ZERO if unwritten

*.

"You"

*.

	Undone	*Redone*	*Neither*
T1	✔		
T2		✔	
T3			✔
T4			✔
T5		✔	

*.

```
Possible answers:
a) Defensive programming
b) N-version programming
c) Retries after a failure
d) Program verification
```

*.

```
A software bug will cause a failure.
```

*.

```
No, since the execution might result in x=1, y=2
```

*.

```
Account 2 equals 3.
```

*.

```
All three copies contain a muddle of old data partially overwritten by
an unknown amount of new data.  There is no assured-to-be-correct source
for any of the data.
```

*.

```
integer procedure Register-two(H1, H2)
        Seize(H1)
        status = Register(H1)
        if (status== 0) then
                Seize(H2)
                status = Register(H2)
                Release(H2)
                if (status == -1)
                        Drop(H1)
                endif
        endif
        Release(H1)
        return status
```

*.

```
The LRU replacement policy updated the database disk with data modified
by an uncommitted transaction, which failed to commit before the crash.
```

*.

	Undone	*Redone*	*Neither*
T1	✔		
T2	✔		
T3	✔		
T4			✔
T5		✔	

*.

```
There are several possibilities.  Here is one simple one:

Write procedure:  write D1, then write D2, then write D3.

Read procedure:
   Read the same block from D1 and D2 and D3
   If all three copies are different, copy D1 to D2 and D3 (completion)
   If the first two copies are the same, copy that value to D3
(completion)
   If the last two copies are the same, copy that value to D1 (backout)
   Return the now-unanimous value
```

*.

To recover to a consistent state after a system crash.

*.

(5)	Second system effect	(1) The Java bytecode verifier
(1), (2), (3)	Leave it to the client	(2) The end-to-end argument
(2), (3)	Keep it simple	(3) Brinch Hansen's RC 4000 (a "microkernel")
(6)	Log updates	(4) 6.033 design project 1
(1)	Static analysis	(5) Multics
(4), (6)	Cache answers	(6) System R

*.

```
Enough to include the last CHECKPOINT, as well as the BEGIN from each
transaction which was uncommitted at the time of the last checkpoint
```

*.

```
For each disk block N
  read block N from D1 and D2 and D3
  If all three copies are different, report unfixable (intolerable
error)
  If any two are the same, copy the agreed-on value over the odd one
  Return the now-unanimous value

The detected intolerable error occurs if we are unlucky enough that the
same block N is corrupted on two or more different drives.  Another
(undetectable) intolerable error occurs if we are unlucky enough that
the same block N is corrupted in exactly the same way on two or three
different drives.  It is interesting that the response to the "all three
copies different" case is not the same as in the answer to part b; the
reconstruction procedure of part c is designed to tolerate a different
set of expected events.
```

*.

```
Account 2 equals 1, 2, or 3 if lines 1 and 2 are switched.
```

*.

A triplex system with repair is sufficient. The MTTF for such a system is approximately the following:

$$\frac{\text{MTTF}}{3 \cdot \text{MTTR}} \times \frac{\text{MTTF}}{2 \cdot \text{MTTR}} \times \text{MTTF} = \frac{1}{6} \cdot \frac{(\text{MTTF})^3}{(\text{MTTR})^2}$$

Plugging the MTTF and MTTR for a single computer gives you a MTTF for the triplex system that is larger than 100 years. Note: duplex with repair is not sufficient, since then the MTTF is approximately the following:

$$\frac{1}{2} \cdot \frac{(\text{MTTF})^2}{\text{MTTR}} = \frac{1}{2} \cdot 10^6$$

*.

Users would no longer have to manually reconcile conflicting changes to the same file.

A file might be unavailable to a user even though the server is reachable and functioning normally, unlike in the current design where files are always available if the server is reachable and functioning.

*.

LFS, since LFS amortizes the high latency cost of seeks by writing complete segments to disk sequentially. As a result, disks can perform at peak disk throughput.

*.

Yes. The value returned is the last value of x that was written by any transaction that committed prior to BEGIN(t). (If there is no such value than 0 is returned.)

*.

```
recover() =
for i := 1 to num-committed do
   t := committed[i]
   for A := 0 to N do /* assume ComRec isn't in 0 to N */
      if exists(A/t) then
         change-tag(A/t, 0)
```

The goal here is to redo the work of trans-commit in case there was a crash in the middle of a trans-commit. Unfortunately, the dirty variable we used in trans-commit doesn't exist any more—it wasn't stored stably. So we conservatively run over the entire disk, looking for Whisk blocks A/t where t is a committed transaction. These blocks contain updates a committed transaction made: we should move them into the stable database state, by changing their tags to 0, just as trans-commit would have.

We don't care about any other transactions (in-progress or aborted); any updates they made are stored in locations with non-zero tags, so don't affect the stable database state even if we don't do anything at all.

*.

A single, global lock which is locked at the start of each transaction and released at COMMIT is atomic, and has no deadlocks.

*.

Given an address A and some new data, we want to run some operations so that the Whisk block A/0 will either have the new data, or its old contents—no partially written stuff—even if the system crashes during the atomic write. We can use the atomic change-tag operation to do this with only two operations:

atomic-write(A, *data*) =
write(A/1, *data*)
change-tag(A/1, 0)

*.

Assuming that the system is using a backout-list log protocol, to the start record of AO42, so that it can be undone. If it is using an intentions-list log (unlikely, because there is no mention of completion records) it would be necessary to go back to the start record of AO7 (which aborted after the checkpoint), to make sure that everything it has changed has been undone.

*.

Improves the performance of workloads dominated by small writes.

*.

```
write(tid, account, newbalance) {
add an entry to the end of intentions list for tid
append WRITE record to log
}

write(tid, account, newbalance) {
append WRITE record to log
add an entry to the end of intentions list for tid
}
```

*.

Read and write performance in the absence of errors is enhanced.

*.

There is no need to write files in place; one can therefore minimize seeks. If the free list is managed in contiguous sections, the next file block write can be sequential. The timestamp distinguishes this block from any old blocks that may be on disk. This is probably faster than LFS because there is no need to store the inodes in the log.

*.

winners: 7, 41, 44, 43, and 46 losers: 42, 45, and 47

*.

Using LFS on the Whisk is not a good idea. LFS was designed to optimize around the high latency (seek time) of a normal disk by making most writes sequential; on the low-latency Whisk, this is unnecessary. Further, LFS buffers writes into large chunks to take advantage of disks' high bandwidth; this will work poorly on the low-bandwidth Whisk. Thus, using LFS on the Whisk has no advantages, and LFS's performance will be worse than that of a "normal" file system! (LFS's failure recovery properties are irrelevant here, because we have given .)

*.
A lock for each variable. Every READ or WRITE operation is immediately surrounded by a LOCK/RELEASE of that variable's lock is not atomic, and has no deadlocks.

*.
The system would recover correctly from the first crash but the log would be corrupt so it might fail to recover correctly from the second crash.

*. It is possible to make the list of losers complete.

*.
System R's storage system (RSS), since it uses a write-ahead logging scheme to support atomic transactions.

*.
Seizing and releasing locks from within subroutines Register and Drop makes it difficult to use those subroutines as components of larger atomic actions such as Register-two. The first clue that there may be a problem is when you notice that the locking sequences that result do not conform to either the simple locking discipline or the two-phase locking discipline. As an example of what can go wrong, suppose that classes 21W731 and 21W732 each have one space available, and Jim requests

Register-two("21W731", "21W732")

at about the same time that Mary requests

Register-two("21W732", "21W731")

Jim takes the last slot in 21W731, then Mary takes the last slot in 21W732, then Jim notices that 21W732 is full, then Mary notices that 21W731 is full. Each then goes on to drop the class they previously registered for, and both go away to look for other classes with space. Although the registrar's data has not been corrupted, this is not an outcome that could have occurred under any serial ordering of these two requests--one of them should have gotten registered in both classes.

*.
14.8 ms.

*.
A lock for each variable. Every variable READ or WRITTEN within a transaction is locked immediately prior to the first reference to that variable in the transaction; all locks are released at COMMIT is atomic and produces deadlocks.

*.
x = 2 Because: AO46 committed that value
y = 4 Because: AO44 committed that value. AO45's attempt to change it has been backed out.

*.

```
abort(tid) {
clear intentions list for tid
}

abort(tid) {
append ABORT record to log
clear intentions list for tid
}

abort(tid) {
clear intentions list for tid
append ABORT record to log
}
```

*.

Initially the three structures are cleared. Then the disk is scanned once and each block is processed. If the block is free (i.e., FID is zero), it is inserted into FREE. Otherwise MAP is probed to see if an entry with the same (FID,SN) exists. If no entry exists, the block is added to MAP. If an entry exists, the block with the older timestamp is added to RECYCLE, and the block with the newer timestamp is inserted into MAP.

Extra: If the block is the first block of a file and the directory FID is zero, the MAP is probed for the other blocks of the file and they are removed from MAP and placed in the RECYCLE list.

*.

Results will correspond to some serial execution of the transactions.

*.

A lock for each variable. All variables READ or WRITTEN within a transaction are locked, in alphabetical order, immediately following the BEGIN. They are all released at COMMIT is atomic and produces no deadlocks.

*.

__a__ 1. training time
__b__ 2. the first system is likely to be quite spartan
__a__ 3. additional communication
__c__ 4. a second system is likely to have better modularity
__a__ 5. repartitioning effort
__b__ 6. each idea looks good in isolation

Some of these causes can underly several effects; the "most likely" ambiguity can be resolved by noting that Brooks identifies certain causes as the principal ones.

*.

A lock for each variable. All variables WRITTEN within a transaction are locked, in alphabetical order, immediately following the BEGIN. They are all released at COMMIT. It is not atomic and produces no deadlocks.

*.

```
trans-commit(t) =
write(ComRec/t, "committed")
for i := 1 to num-dirty do
   change-tag(dirty[i]/t, 0)

trans-abort(t) =
write(ComRec/t, "aborted")
/* we don't have to worry
     about the garbage! */
```

Trans-commit writes the commit record, and moves all of transaction t's updates (in blocks dirty[i]/t) into the stable database state by changing those blocks' tags to 0. Note that the commit record **must** be written **before** the updates are moved. Imagine that you moved the updated blocks first. Then, if a crash happened in the middle of moving updates, the recovery procedure would have no way of distinguishing the following situations, since in both cases, the commit record would just read "in progress".

1. The transaction had not yet committed. It must be aborted; its updates should be thrown away.
2. The transaction was in the middle of committing. Some of its updates have made it to the stable database, so its remaining updates must be redone.

*.

The Coda file system, since it uses optimistic concurrency control and file hoarding to allow clients to operate while disconnected from the file server.

*.

The only shared variables are X and Y, so they must be the clue. Reconstructing their version history up to the point where AO_{45} writes Y, we run across the following remarkable state:

AO	7	41	42	43	44	45	46
X				5->9			
Y				5->6	6->4	4->3	
commit state	A	U	U	C	U	U	U

AO_{45} is using a value for Y that AO_{44} has changed, but has not yet committed. If AO_{44} aborts, AO_{45} will also have to abort. The evidence is ambiguous because there might be an incredibly complicated, optimistic coordination rule that includes checking when AO45 tries to commit that AO44 didn't change Y again and that AO44 actually committed.

*.

So that the recovery manager knows what to redo.

*.

Customer complaints

*.

Only one block write is required. All of the information about the file is contained in the first block of the file.

*.

xFS, since it exploits the fast network to do cooperative caching, resulting in very high performance.

*.

Thus pass the earthly glories.

*.

```
III. a-c, as in protocol I.
     d.  Everyone, if their calendar is still open at the chosen time
            Send Pat an acknowledgement.
         Otherwise
            Send Pat apologies.
     e.  Pat collects the acknowledgements.  If all are positive
            Send a message to everyone saying the meeting is ON.
         Otherwise
            Send a message to everyone saying the meeting is OFF.
     f.  Everyone, if they received the ON/OFF message,
            acknowledges receipt.
         Otherwise
            sends a message to Pat asking what happened.
```

*.

Step d tentatively commits the individual participants, but because some participants may send apologies rather than commitments the operation as a whole cannot commit until Pat knows that everyone has accepted. Step e is therefore required, and the sending of Pat's ON/OFF message is the commitment step for the operation.

Steps g and h are a refinement that allow Pat to discard her notes after the last acknowledgement comes back, but they contribute nothing to the 2-phase commit protocol itself.

*.

It is faster to read the buffer cache than to read the disk.
If common access patterns can be identified, performance can be improved by prefetching multiple account balances into the cache.
It reduces the total number of disk reads when one transaction reads the same account balance multiple times without updating it.
It reduces the total number of disk reads when multiple consecutive transactions read the same account balance.

*.

It doesn't introduce deadlocks, but doesn't guarantee serialized atomic transaction semantics either.

*.

AFS, since it caches files on the client's disk to hide network latency, and provides authentication and authorization.

* .

Operation	**Write the following record(s) to the log**	**Write the following record(s) to the database**	**Write log or database first?**
Start transaction	*BEGIN N*		
Abort transaction	*ABORT N (optional)*	*DATA to undo all changes*	*Database*
Commit transaction	*COMMIT N*		
Read variable			
Write variable	*UNDO N*	*DATA*	*Log*

 Name ______________________