DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033—Computer System Engineering Handout 33—May 13, 1996

Solutions to Quiz 3 Sample Questions

Here are the solutions to the sample questions given out last Wednesday. Good luck with the quiz on Wednesday. If you've been missing everything, here are the details given on the quiz announcement:

1. Quiz 3 will be from **2–3pm** on **Wednesday, May 15, 1996.**

2. There are ***three*** rooms. Go to the room that corresponds to your last name:

   4-270 <---> A to H
   4-370 <---> I to R
   2-190 <---> S to Z

3. The quiz will cover all the material from Lecture 17 to Recitation 24.

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

Below are the solutions from the previous two years. Have fun and good luck.

Table of Contents

1. Spring 1995 Quiz 3 Solutions.
2. Spring 1994 practice question solutions.

1. *(20 points)* You are designing a computer system to control the power grid for the Northeastern US. If your system goes down, the lights go out and civil disorder —riots, looting, fires, etc.—will ensue. Thus, you have set a goal of having a system MTTF of at least 100 years (about 10^6 hours).

1a. For hardware you are constrained to use a building block computer that has a MTTF of 1000 hours and a MTTR of 1 hour. Assuming that the building block is self checking (i.e., fail stop), how can you arrange to meet your goal? (Briefly explain your answer.)

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

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

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:

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

1b. Your software is 10^6 lines of code. Even after rigorous testing, the software can be expected to have about one bug per 10^3 lines or about 10^3 bugs. You may assume that these bugs are unrelated. Discuss briefly a strategy for how to arrange your software to meet the goal of a MTTF of at least 100 years.

Possible answers:
a) Defensive programming

b) N-version programming

c) Retries after a failure

d) Program verification

2. *(30 points)* The registrar's office is upgrading its scheduling program for limited-enrollment humanities subjects to make it a network service that anyone can use by RPC. But there is a concern that having multiple threads trying to update the database at the same time could possibly cause trouble.

The program operates as follows; the interface originally had just two entries:

```
Status = Register(subject-name)
```

and

```
Drop(subject-name)
```

where subject-name is a string such as "21W471". The Register procedure checks to see if there is any space left in the subject, and if there is, it increments the class size by one and returns the status value zero. If there is no space, it does not change the class size; instead it returns the status value -1. (This is a very primitive registration system—it just keeps counts!)

As part of the upgrade, two new procedures have been added:

```
Seize(subject-name)
```

and

```
Release(subject-name)
```

Seize sets a lock on the given subject, waiting if necessary for the lock to be released.
Release releases the lock on the given subject.
Here is a typical application program, which registers the caller for two subjects, H1 and H2:

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

2a. The goal is that the entire procedure Register-two should be an atomic transaction. Add calls for Seize and Release to the Register-two procedure that obey the Simple Locking Rule (i.e., Seizes before the first read/write, and Releases after the last read/write).

```
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
```

2b. Add calls for Seize and Release that obey the Two-phase Locking Rule, and in addition postpone all Seizes as late as possible and do all Releases as early as possible.

```
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
```

2c. Louis Reasoner has come up with a suggestion that he thinks could simplify the job of programmers creating atomic application programs such as Register-two. His idea is to revise the two programs Register and Drop by moving the Seize and Release statements into them. That is, the procedure:

```
integer procedure Register(subject-name)
        { current code }
        return status
```

would become:

```
integer procedure Register(subject-name)
        Seize(subject-name)
        { current code }
        Release(subject-name)
        return status
```

As usual, Louis has misunderstood part of the problem. Give a brief argument of what is wrong with this idea.

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.

3. *(20 points)* Consider a database system in which all transactions are atomic with respect to system failures and other, concurrent, transactions. Locks on data records along with a log in nonvolatile memory are used to provide these properties. The database itself is stored on disk and cached in main memory by the cache manager. The database system is able to write the following records to the log:

   - BEGIN N: start of transaction number N
   - COMMIT N: commit of transaction N
   - ABORT N: abort of transaction N
   - UNDO N: a record to be able to undo part of transaction N
   - REDO N: a record to be able to redo part of transaction N

   In addition, the system writes DATA records to the on-disk database.

3a. Suppose the cache manager implements a write-through policy. That is, whenever a transaction writes a record to disk to update the database, the record is synchronously forced to disk. Explain what minimum records need to be written to the database and the log, and in what order for the following database operations: start transaction, abort transaction, commit transaction, read of variable, and write of a variable?

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*

3b. In the system from part (a), how is the state of the system recovered from the log after a system failure (i.e., contents of main memory are lost)?

The log is scanned forward to identify uncommitted transactions (active when the system failed). The log is then scanned backwards and all the UNDO records for the uncommitted transactions are processed to undo all actions of any uncommitted transaction. If the design provides a way of finding the end of the log without having to scan forward from the beginning, then it is possible to do the entire job with a single reverse log scan, collecting the list of uncommitted transactions as you go.

4. *(30 points)* Ben Bitdiddle has proposed the simple neat and robust file system (SNARFS). Ben's system has no on-disk data structures other than the disk blocks themselves. Each 4K-byte disk block starts with the following 24 bytes of information:

   - FID (File-ID): a 64-bit number that uniquely defines a file. A FID of zero implies that the disk block is free.
   - SN (Sequence Number): a 64-bit number that identifies which block of a file this disk block contains.
   - T (Time): The time of last update.

   In addition, the first block of a file contains the file name (string), version number, and the FID of its parent directory. The rest of the first block is filled with data. Setting the directory FID to zero marks the entire file free.

   Directories are just files. Each directory need contain only the FID of its parent directory. However as a 'hint' directories may also include a table giving the mapping from name to FID and the mapping from FID to blocks for some of the files in the directory.

   To allow fast access, three in-memory (virtual memory) structures are created each time the system is booted:

   - MAP: an in-memory hash-table that associates a (FID,SN) pair with the disk block containing that block of that file.
   - FREE: a "buddy-system" free list that represents all of the free blocks on disk in a compact manner. (A buddy-system maintains a list of contiguous regions of free disk space.)
   - RECYCLE: a list of blocks that are available for reuse but have not yet been written with an FID of 0.

4a. What is the minimum number of disk I/Os required in Ben's SNARFS system to create a file in an existing directory, and write a 2K-byte string into the file? If the system crashes (i.e., the contents of virtual memory are lost) after these I/Os are completed, the file should be present in the appropriate directory after recovery. Each read or write of a disk block results in one disk I/O. (Briefly explain your answer.)

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

4b. Ben argues that the in-memory structures can be rebuilt after a crash. Explain briefly what actions are required to rebuild MAP, FREE, and RECYCLE at boot time.

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.

4c. Ben also argues that with appropriate free list management, the write performance of SNARFS can be as fast as for a log-structured file system. How can this be?

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.

Problem 1. *(20 points)* You are examining a 3.5-inch magnetic disk that spins at 7500 RPM, with an average seek time of 6 milliseconds and a data transfer rate of 2 Megabytes/second.

a. How long does it take to read a block of 4 Kilobytes chosen at random?

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.	

Suppose that the operating system maintains a one-Megabyte cache in RAM that hold disk blocks. The access time to this cache is 125 nanoseconds and the data transfer rate is 20 Megabytes/second.

b. Give the formula that tells the average time to read a 4 Kilobyte disk block, assuming that the hit-ratio of the disk block cache is h.

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.

c. Ben Bitdiddle's workstation has 24 Megabytes of RAM. To increase h, Ben reconfigures the disk block cache to be much larger than one Megabyte. To his surprise he discovers that many of his applications now run slower rather than faster. What has Ben probably overlooked?

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.

d. Louis Reasoner has gone over to the Lab for Computer Science and has disassembled your disk unit to see how it works. Remembering that the centrifuge in the biology lab runs at 36,000 RPM, he has come up with a bright idea on how to reduce the rotational latency of the disk. He suggests speeding it up to 96,000 revolutions per minute. He calculates that the rotation time will now be 625 microseconds.

One of the other UROP's in the lab says this idea is crazy. Explain the line of reasoning that the other student probably has in mind.

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.

Problem 2. (18 points)

Your company has asked you to take the responsibility for developing a storage system for a very demanding application, so you have decided to use a triplexed stable-storage scheme. You will use three ordinary disk drives D1, D2, and D3, and arrange that D2 and D3 store identical copies of each block stored on D1. The disk drives are of a very simple design that does not report read errors. That is, they just return data, whether or not it is valid.

a. You initially construct the application so that it writes a block of data to all three drives concurrently. After a crash occurs during the middle of a write, you are unable to reconstruct the correct data for that block. What is the problem?

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.

b. Describe a simple modification to solve this problem.

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
```

c. One day there is a really awful power glitch that crashes all three disks in such a way that each disk corrupts one random track. Fortunately, the system wasn't writing any data at the time. Describe a procedure for reconstructing the data. Explain any cases that your procedure cannot handle.

```
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.

3. *(20 points)* A system that implements atomic operations writes the following types of records to a log maintained on stable storage:

<AO_i ***start***>	Atomic operation *i* starts.
<AO_i, *x, old, new*>	Atomic operation *i* writes the value *new* over the value *old* for the variable named *x.*
<AO_i ***commit***>	Atomic operation i commits.
<AO_i ***abort***>	Atomic operation i aborts.
<AO_i, AO_j, … AO_k; ***checkpoint***>	At this instant, atomic operations i, j,…k are uncommitted.

Atomic operations start in numerical order. A crash occurs, and the recovery procedure scans the log backwards to the latest checkpoint, finding the following records:

<AO_7, AO_{41}, AO_{42}; ***checkpoint***>
<AO_{43} ***start***>
<AO_{44} ***start***>
<AO_{43}, y, 5, 6>
<AO_{43}, x, 5, 9>
<AO_{43}, ***commit***>
<AO_{44}, y, 6, 4>
<AO_{45} ***start***>
<AO_{45}, z, 3, 4>
<AO_7 ***abort***>
<AO_{41}, q, 1, 9>
<AO_{46} ***start***>
<AO_{45}, y, 4, 3>
<AO_{44} ***commit***>
<AO_{45}, y, 3, 7>
<AO_{41} ***commit***>
<AO_{47} ***start***>
<AO_{46} x, 9, 2>
<AO_{46} w, 0, 1>
<AO_{46} ***commit***>
<AO_{47}, u, 2, 1>
(***** crash happened here *****)

a. How much farther back in the log should the recovery procedure scan?
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.

b. Which atomic operations mentioned in this part of the log are *winners* and which are *losers*?
winners: 7, 41, 44, 43, and 46 losers: 42, 45, and 47

c. Is it possible to make the list of losers complete with just the information given? Yes/No

d. What are the values of x and y immediately after the recovery procedure finishes? Why?
x = 2 Because: AO46 committed that value
y = 4 Because: AO44 committed that value. AO45's attempt to change it has been backed out.

4. *(5 points)* The log of problem 3 contains (ambiguous) evidence that someone didn't follow coordination rules. What is the evidence?

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.

5. *(5 points)* The intentions-list logging protocol requires that the commit (or abort) record should be written to the log BEFORE any updates to cell storage are made. Identify the best reason for this requirement.

_____ 1. So that the recovery manager knows what to undo.
__X__ 2. So that the recovery manager knows what to redo.
_____ 3. Because the log is less likely to fail than the cell storage.
_____ 4. To minimize the number of disk seeks required.

6. *(12 points)* Brooks describes several interesting effects, three of which are listed below:

a. The mythical man-month.
b. The second-system effect.
c. Plan to throw one away.

Following are several root causes. Indicate beside each one the letter of the one effect it is most likely to cause.

__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 underlie several effects; the "most likely" ambiguity can be resolved by noting that Brooks identifies certain causes as the principal ones.

7. Pat, Diane, and Quincy are having trouble using e-mail to schedule meetings of their 6.033 case study group. Pat suggests that they take inspiration from the 2-phase commit protocol.

(15 points) Which of the following most closely resembles 2-phase commit? (Circle the appropriate Roman numeral.)

```
  I. a.  Pat requests everyone's schedule openings.
     b.  Everyone replies with a list but does not guarantee to hold all the
            times available.
     c.  Pat inspects the lists and looks for an open time.
         If there is a time,
            Pat chooses a meeting time and sends it to everyone.
         Otherwise
            Pat sends a message canceling the meeting.

 II. a-c, as in protocol I.
     d.  Everyone, if they received the second message,
            acknowledges receipt.
         Otherwise
            sends a message to Pat asking what happened.

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.

 IV. a-f, as in protocol III.
     g.  Pat sends a message telling everyone that everyone has confirmed.
     h.  Everyone acknowledges the confirmation.
```

(5 points) Below, circle the letter of the step that commits the meeting time.

a b c d (e) f g h

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.