DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033—Computer System Engineering Handout 36—May 18, 1995

Quiz #3 Solutions

```
Statistics gathered from 171 exams:
Median score = 57
Mean score = 58.5
Std. deviation = 17

ASCII Histogram:

Score  Frequency
16-20  ** (2)
21-25  *** (3)
26-30  **** (4)
31-35  ****** (6)
36-40  ******* (7)
41-45  **************** (16)
46-50  ******************* (19)
51-55  ********************* (21)
56-60  *************** (15)
61-65  *********************** (23)
66-70  ********** (10)
71-75  ************ (12)
76-80  ************** (14)
81-85  ********* (9)
86-90  ***** (5)
91-95  **** (4)
96-100 * (1)
```

For Official Use Only

Problem	Your Score	Maximum Score
1		20
2		30
3		20
4		30
Total		100

Name: ______________________________

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

Name: ______________________________

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
```

Name: ______________________________

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
```

Name: ______________________________

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.

Name: ______________________________

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.

Name: ______________________________

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.