DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

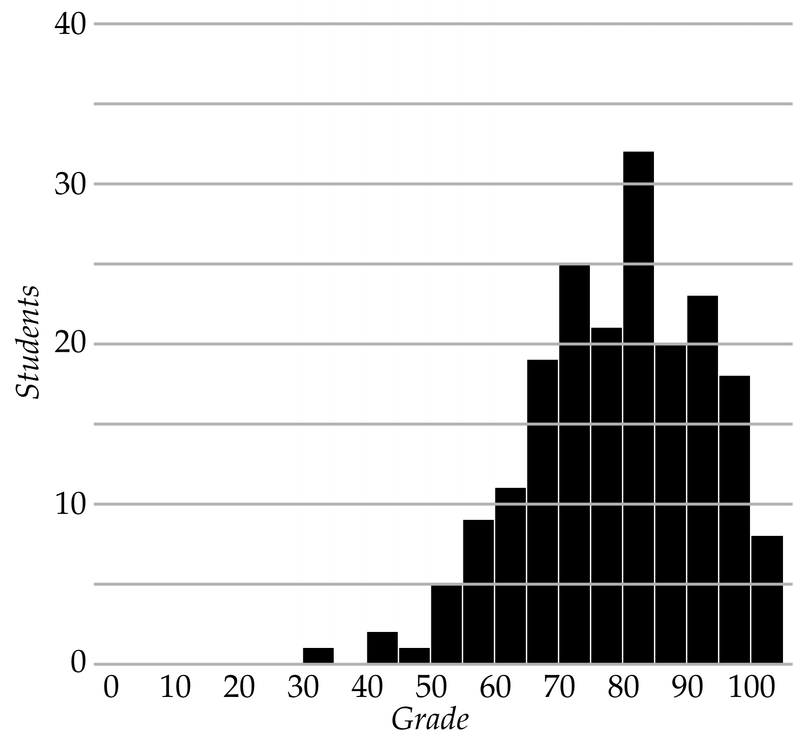
6.033—Computer System Engineering — Handout 37—May 16, 1996

Quiz 3 Solutions

This quiz contains five problems; we have written our solutions in the space provided.

Check the 6.033 Home Page for information on how to obtain final grades.

Grade distribution



Average 80 *Standard deviation* 14

Name: *Dennis Ritchie (I think)*

1. *(20 points)* Here is a list of 5 different storage systems.

   (1) RSS, the System R storage system
   (2) A log-structured file system (LFS)
   (3) xFS
   (4) AFS (Athena's file system)
   (5) The Coda file system

   For each situation below, choose the storage system that would be best suited for that situation, and briefly explain your answer. For each situation, you should consider the workload placed on the storage system; consider any special requirements for scalability, security, or reliability; and maximize speed. You will use each storage system only once.

1a. *(4 points)* A file server with a write-dominated workload

> ***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.*

1b. *(4 points)* Transaction processing

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

1c. *(4 points)* A file server for a large network where clients are often disconnected

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

1d. *(4 points)* A very fast file server for a small, fast network with trusted clients

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

1e. *(4 points)* A file server for a large, untrusted, and relatively slow network

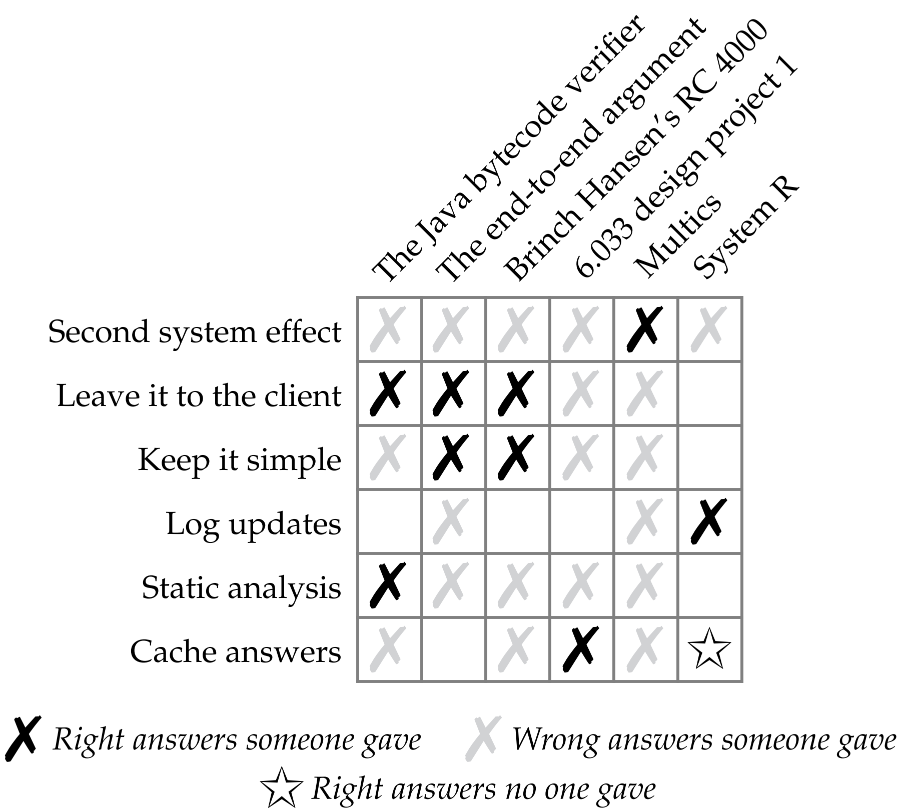
> ***AFS***, *since it caches files on the client's disk to hide network latency, and provides authentication and authorization.*

2. *(18 points)* Match each "hint for system design" (from Lampson's paper) or effect (from *The Mythical Man-Month*) with a system or argument that exhibits that effect, or uses that hint.

Here are all the possible correct answers:

(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

And just for fun, here's a plot of all the answers students actually gave!



3. *(28 points)* Alyssa P. Hacker needs a transaction-processing system for remotely updating information about her collection of 97 cockroaches. Having taken 6.033, she remembers System R. However, she thinks that System R was much too complex, and she can do better.

3a. *(4 points)* First, Alyssa stores her database on disk. When a transaction commits, it simply goes to the disk and writes its changes in place over the old data.

Give one major problem with Alyssa's system.

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?)

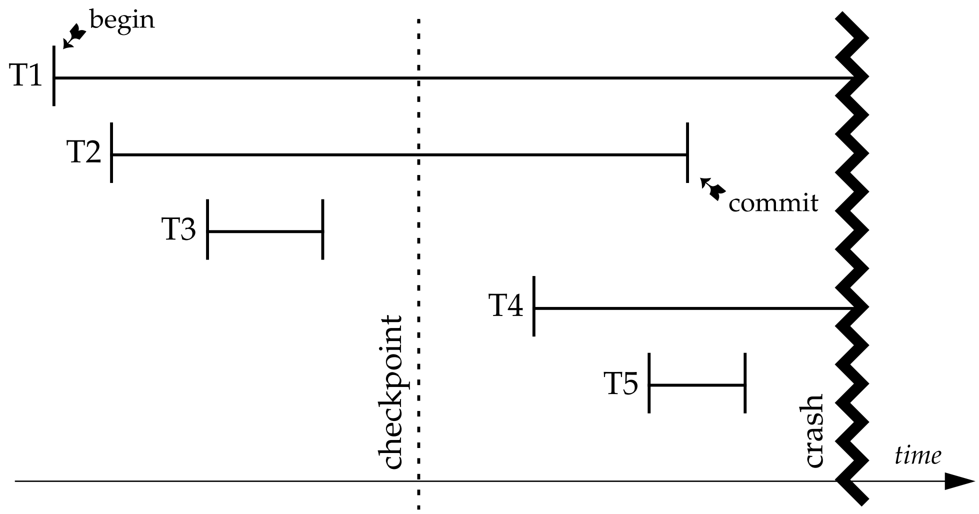
3b. *(4 points)* In Alyssa's second design, the *only* structure she keeps on disk is a log. This log records every change made to the database, along with the transaction which the change was a part of. Commit records, also stored in the log, indicate when a transaction commits. When the system crashes and recovers, it replays the log, redoing each committed transaction.

Give one disadvantage of Alyssa's system.

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.

Now, Alyssa decides to change her system by storing both the log, and a snapshot of her database. She remembers that System R called this snapshot mechanism a *checkpoint*. To checkpoint, Alyssa just writes the entire state of the database into the log. When she crashes and recovers, she starts from the last checkpointed state, and then redoes or undoes some transactions to restore her database.

The other parts of this question refer to these five transactions:



So T2, T3, and T5 commit before the crash, but T1 and T4 have not committed by that time.

3c. *(10 points)* When the system recovers, after the checkpointed state is loaded, some transactions will need to be undone or redone using the log. For each transaction, mark off in the table whether that transaction needs to be undone, redone, or neither.

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

3d. *(10 points)* Now, assume that transactions T2 and T3 were actually *nested* transactions: T2 was nested in T1, and T3 was nested in T2. Again, fill in the table:

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

4. *(34 points)* ODC (an acronym for Odd Disk Company) has just invented a new kind of stable storage, the Whisk. A Whisk is unlike a disk in the following ways:

- Whisks have very low read and write latencies. (Disk latency is high for both operations.)
- On the other hand, a Whisk's bandwidth is much less than that of a disk.
- Whisks are *associative.* A block is named with a pair of items: an address and a *tag*. (Disks just use addresses.) We write these pairs as A/*t*, where A is the address and *t* is the tag. Thus, for example, there might be three blocks on the Whisk with address 49, if those three blocks had different tags (e.g., 49/1, 49/2, and 49/97).

The Whisk provides four important operations:

- *read*(A/*t*) → *data*. This is the normal read operation.
- *write*(A/*t*, *data*). Just like a normal disk, if the system crashes during a *write*, a partially written block may result.
- *exists*(A/*t*) → *boolean*. This returns true if block A/*t* exists on the Whisk.
- *change-tag*(A/*m*, *n*). This operation is new: *change-tag*(A/*m*, *n*) **atomically** changes the tag *m* of block A/*m* to *n* (deleting any previous block A/*n* in the process). If the system crashes during a *change-tag*, the disk will not be left in a partially-changed state.

Ben Bitdiddle is very excited about the properties of Whisks! Help him develop different storage systems using Whisks as the medium.

4a. *(7 points)* At first, Ben emulates a normal disk by writing all blocks with tag 0. He wants to add an atomic write operation to this system. Give the operations required to atomically write a block on Ben's Whisk, and circle the operation that is the commit point. (In class, we gave an atomic-write algorithm for disks which required four operations; you should be able to do it with less.)

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*)
  $\boxed{\textit{change-tag}(\text{A}/1, 0)}$

4b. *(5 points)* Now, Ben is considering implementing a log-structured file system with the Whisk. Is this a good idea? Consider the properties of the disk LFS was designed for.

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 .)

Now, Ben is working on a Whisk transaction system; he'd like you to help him finish it. Looking through his notes, you see that Ben's system *will use no caches or logs*: all writes go straight to the Whisk. One sentence particularly catches your eye: a joyfully scrawled "***Transaction IDs Are Tags!!***"

Ben's basic idea is this. The current state of the database will be stored in blocks with tag 0. When a transaction *t* writes a block, the data is stored in the separate block A/*t* until the transaction commits.

Ben has set aside a special disk address, **ComRec**, to hold commit records for all running transactions. For a valid transaction ID *t,* the contents of **ComRec**/*t* is either "committed", "aborted", or "in progress", depending on what state *t* is in.

So far, begin, read, and write have been implemented. (Below, *t* is a transaction ID, A an address, and *data* a data block.)

```
trans-begin(t) =
  write(ComRec/t,
        "in progress")

trans-read(t, A) =
  if exists(A/t) then
    return read(A/t)
  else /* haven't written yet */
    return read(A/0)

trans-write(t, A, data) =
  write(A/t, data)
```

In parts c and d, you are asked to write pseudocode for commit and abort, and design a system for recovery after a crash. You may want to look through both parts before beginning. **Do not worry about concurrent transactions:** we are only concerned with atomicity in the face of failure.

4c. *(12 points)* Write pseudocode for *trans-commit*(*t*) and *trans-abort*(*t*), and circle the commit point in *trans-commit*. You can assume that the variable *dirty*, an array with *num-dirty* elements, holds all the addresses to which *t* has written. (Don't worry about any garbage an aborted transaction might leave on disk, and assume transaction IDs are never reused.)

```
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.

4d. *(10 points)* How would you implement recovery after a crash? Ben has already done some of the work: his code examines the **ComRec** blocks and determines which transactions are committed, aborted, or in progress. When your pseudocode is called, he has already set 6 variables for you (you might not need them all):

num-committed	the number of committed transactions
committed	an array holding the committed transactions' transaction IDs
num-aborted	the number of aborted transactions
aborted	an array holding the aborted transactions' transaction IDs
num-in-progress	the number of transactions in progress
in-progress	an array holding the in-progress transactions' transaction IDs

(*Hint:* You may assume that addresses, A, run from 0 to *N*.)

```
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 in locations with non-zero tags, so they don't affect the stable database state anyway.

5. ***EXTRA CREDIT!!*** *(1 point)* Circle Ken Thompson. *Here are three correct answers!*