DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033—Computer System Engineering Handout 34—May 15, 1996

Quiz 3

This quiz contains five problems whose relative weights are given below. Please write your answers in the space provided, and hand in this booklet at the end of the examination.

Put your name on this cover sheet *and* at the bottom of each page of this booklet.

Some parts of some problems may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. Some problems are intentionally ambiguous; be sure to write down any assumptions you make. Show your work, or you risk losing partial credit. Be neat. If we can't figure out your answer we can't give you credit.

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

Circle your recitation:

10:00	1. Shrira/Litt			
11:00	7. Corbató/Fu	5. Ganger/Yerushalmi	12. Shrira/Kohler	2. Wetherall/Legedza
12:00	6. Wetherall/Yerushalmi			
1:00	3. Agarwal/Legedza	11. Corbató/Litt	9. Ganger/Fu	8. Saltzer/Engler
2:00	4. Agarwal/Kohler	10. Saltzer/Yerushalmi		

For Official Use Only

Problem	*Your Score*	*Maximum Score*
1		20
2		18
3		28
4		34
5		1
Total		101

Name: _____

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

1b. *(4 points)* Transaction processing

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

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

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

Name: _____

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.

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

Name: _____

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.

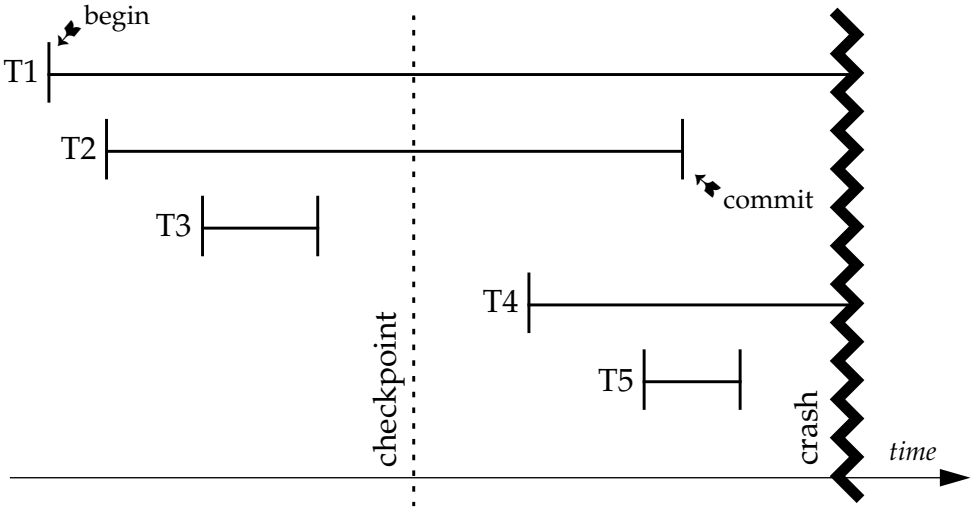
3b. *(4 points)* In Alyssa's second design, the *only* structure she keeps on disk is a log. This log records every change made to the database, along with the transaction which the change was a part of. Commit records, also stored in the log, indicate when a transaction commits. When the system crashes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

Give one disadvantage of Alyssa's system.

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.

Name: _____

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			

Name: _____

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.)

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.

Name: _____

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.)

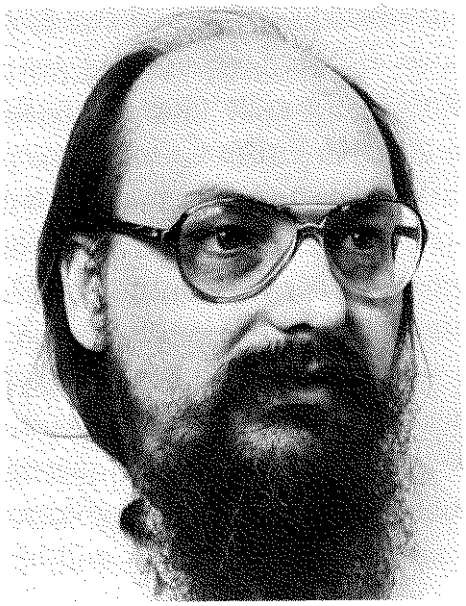
Name: _____

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*.)

5. ***EXTRA CREDIT!!*** *(1 point)* Circle Ken Thompson.



Name: _____