Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 33 October 27, 1993

Problem Set #7

Due date: November 3, 1993

Problem 1 (50 points)

The abstraction function given in Handout 31 for the `LogAndCache` implementation of `SequentialTransactions` is wrong: the value given for `vs` doesn't work. The third invariant is similarly wrong. Your job in this problem is to fix them. To that end, you should:

(A) Give a correct abstraction function for `LogAndCache`. Fix the problem with the existing abstraction function for `vs`, and also give the abstraction function for `ph`.

(B) Fix the third invariant to explain correctly how `sb++c` relates to `sb`, `sl`, `sul`, `vl` and `vul`.

(C) Argue informally (but convincingly) that the abstraction function you gave in part (A) is correct. Identify the atomic actions in `LogAndCache`, and argue briefly for each how it is explained by an appropriate sequence of atomic actions in `SequentialTransactions`. (Note: this may seem like a big job, but there aren't actually that many atomic actions to be considered. Feel free to apply the techniques of Handout 21 to reduce the number of actions even further.)

Hint: as discussed in lecture, `vs` should include all the effects of the currently active transaction. Since the volatile log is forced incrementally in the background (by `Flush`) to the stable log, the abstraction function given in Handout 31 fails to account for the updates that have already been written to stable storage (since `ss` doesn't change until `Commit`). To figure out a correct abstraction function, you will need to enumerate all the atomic actions in the implementation and decide how each should be explained in terms of the specification. Then you need to come up with an expression for `vs` that makes your explanation work. For example, unlike with the expression for `vs` given in Handout 31, the background actions of forcing or truncating the log of flushing the cache should have no effect on `vs`.

Hint: notice that in the specification, the `Do` action chooses the result of an action by running it in `vs`, while in the implementation it runs the action in the state `sb++c`. For this to work, `vs` must be `sb++c` whenever `Do` can happen. However, simply defining `vs` to be `sb++c` does not work; there are times when a more complex expression is needed.

Hint: you may find it convenient to define some history variables to keep track of additional state. If you do so, be sure to show how the code is modified to update the history variables, and also be sure to state the invariants that relate the history variables to the rest of the state components.

Problem 2 (50 points)

In this problem you are to develop an "optimistic" implementation of the `ConcurrentTransactions` specification in Handout 32. Your implementation should never delay a `Do` operation; instead, it should check when a transaction attempts to `Commit` whether committing the transaction will preserve the invariant. To simplify the problem, you may use the `NC` variant of the invariant.

Assume that a state `S` is structured as a sequence of independent components, and that actions can be classified as either reads or writes. (If you want, assume a method `kind` on `A` that tells you whether a given action is a read or a write.) Your solution should have the property that a transaction can always commit unless some other transaction conflicts with it while it is running (two transactions conflict if both access the same component and one of them writes the component.)

Your main task in this problem is to figure out what data structures to maintain so that you can check for conflicts during `Commit`. You should not worry about low-level synchronization, so you can simply write each of the external operations (`Begin`, `Do`, `Commit`, and `Abort`) as `APROC`s.

You should also not worry about making recovery from `Abort`s efficient; for example, when a transaction writes a component, you could just keep both the old value and the new value around. Be careful, though: if you update a component with a new value and then restore an old value when a transaction aborts, you could lose other committed updates if some other concurrent transaction wrote the same component and then committed. So you might be better off not installing updates in the "base copy" until commit.