Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 48 November 24, 1993

Solutions to Problem Set #9

The maximum score for this problem set was 150 points.
The problem statements are not included here, please refer to Handout 41, Problem Set #9.

Problem 1 (20 points)

(A) (5 points)

```
RI(i) = i.last = 0 /\ (ALL j | i(j) >= 0 /\ i(j) < MaxB)
AF(i) = i.size > 1 => RET i.head + 256*AF(i.tail) [*] RET 0
```

(B) (5 points)
Yes — every behavior produced by `SimpleInt` is allowed by `BigInt`; the fact that `BigInt` represents an integer using a sequence of values ranging from 0 to 255 is not visible outside the module. A correctness proof showing that `SimpleInt` implements `BigInt` would require a history variable, since `BigInt` does not garbage-collect trailing zeroes in the sequence.

(C) (5 points)
No — `Dec` and `IsZero` will yield `HAVOC`, which cannot be mapped into any part of the specification.

(D) (5 points)
Yes — the abstraction function simply maps everything to `nil`; all exceptions and return values are covered by `HAVOC`.

Problem 2 (12 points)

(A) (4 points)
The time is 10 ms to seek, $60/5400 = 1/90 = .011$ sec for a full rotation, so .0055 sec on average for half a rotation, and 1 $MB/3MB = 333$ ms transfer time, or about 350 ms.

(B) (4 points)

For each block, $15.5\ ms$ overhead plus $10K/3M = 3.33\ ms$ transfer time $= 18.8\ ms$ total, times 100 blocks, $= 1.88\ sec$. It's $1.88/.35 = 5.4$ times slower.

(C) (4 points)

$1.1 * 350 = 385$. Subtracting the transfer time of 333 yields $52\ ms$ for access time. Each access costs $16.5\ ms$, so we can afford $52/16.5 = 3.15$ accesses. Hence the block size should be 333 KB.

Problem 3 (6 points)

We encode the function as a sequence of `(N, IN)` pairs, preceded by an encoding of the number of pairs. The order is not important. We assume that sequences of encoded `(N, IN)` pairs can be parsed unambiguously, and maintain that property for `Dir` by prefixing the length of the encoding.

```
PROC EncodeDir(dir: N -> IN) -> SEQ Byte = VAR s: SEQ Byte |
    DO VAR n | dir!n =>
        s := s + n.enc + dir(n).enc; dir:= dir{n -> }
    OD; RET s.size.enc + s
```

Problem 4 (12 points)

(A) (6 points)

```
FUNC dir() -> (N->SEQ BYTE) = (LAMBDA (n) -> SEQ Byte =
   VAR in := dir(n).inode,
       i := 0,
       f: SEQ Byte := {} |
     DO i < in.size =>
       VAR db | db IN disk /\ db.inode = in /\ db.blockno = i/dbSize =>
         f := f + db.data; i := i + dbSize
     OD;
     RET f.sub(0, dir(n).size-1)
```

Note that the hash structure has nothing to do with the abstraction function.

(B) (6 points)

It's terrible for sequential access, and not so great even for random access when the disk gets full, because it takes one disk access to follow each link of the hash chain.

Problem 5 (12 points)

(A) (6 points)

Looking backward from an "elect `a`" chunk to the node with address `a`, all the nodes you pass have addresses `> a`. Note that there can be several Elect chunks, and that the index of a node in the ring is not the same as its address. It's sufficient to have this invariant only for the smallest address, but that's harder to write.

```
( ALL n0, n1, e: Elect | z!n0 /\ z!n1 /\ e IN z(n1).b /\ z(n0).a = e.a
==>
  (ALL n | Between(n, n0, n1) ==> z(n).a > e.a) )
```

(B) (6 points)

Let the maximum time for a chunk to traverse the ring by $tmax$, the maximum number of nodes be $nmax$, and the maximum time to transmit a packet be $pmax$. Since each node sends only one packet each time it gets the token, the maximum time it can take the token to loop around the ring is $tmax + pmax * (nmax - 1)$ so this is a suitable timeout since the last time you saw the token. Alternatively, if a node sees only "idle" chunks for $tmax$, the token must be lost.

Problem 6 (12 points)

(A) (6 points)

The key point is that the log idempotence property does not follow from the weaker update idempotence property. For example, given arbitrary updates, we can give them the update idempotence property by recording in the state the last update to have been done, and changing each update so that it does nothing if it was just done. Now consider two updates `u1` and `u2`, where `u1` increments `x` by 1 and `u2` increments it by 2. If we make `u1` and `u2` update-idempotent, the sequence `u1`, `u2`, `u1`, `u2` has the effect of incrementing `x` by 6, whereas if they had the log idempotence property, it would increment `x` only by 3.

(B) (6 points)

After applying `l.head` to `ss`, `Redo` should set `sl` to `sl.tail`. This ensures that an update is deleted from the stable log after being applied to the stable state and before any other update is applied. If a crash occurs, the update at the head of the stable log will either be the last update applied or will not have been applied at all. Note: this is not particularly efficient.

Problem 7 (12 points)

(A) (6 points)

If each action can be done as an atomic update, then the only change is to `Commit`, which must make sure that all of a transaction's updates are stable before committing the transaction. (Recall the "redo rule" from class.) The `LogAndCache` implementation in Handout 31 ensures this by forcing the redo log. Without a redo log, the only way to ensure this is to flush the cache before writing the commit record. (Flushing the cache must still obey the write-ahead-log rule, so we can undo things if there's a crash before the commit.) Note that we must also require that the undo actions can be done as a single atomic update to stable storage.

A number of people decided to apply the updates in `vl` to `sb`; this works, but it's easier and simpler to eliminate `vl` along with `sl`, and just make sure the cache is flushed before committing the transaction.

Several people also wanted to enforce the constraint that updating `sb` for each action would happen atomically with moving an undo from `vul` to `sul` (a natural thing to attempt if you simply think of updating `sb` as playing the role of moving updates from `vl` to `sl`). This is a bad idea. It's okay to write `sul` ahead of `sb`, as long as you can tell (e.g., using tags in the state) whether an undo needs to be applied.

Note that the updates need to be flushed to `sb` by the time `Commit` returns, but they do not need to be done synchronously with `Commit` or with `Do` – they can still be flushed in the background.

(B) (6 points)

A lot of random I/O may be required to commit a transaction, either during the commit operation itself or while the transaction is running. Also, undo's must be logged as physical (atomic) updates rather than logical actions, so they will take more space and also less concurrency will be possible.

Problem 8 (12 points)

(A) (6 points)

The minimum packet size must be at least the number of bytes that can fit in a single network round-trip. So the maximum diameter of the network is limited by the minimum packet size.

(B) (6 points)

The addresses used in messages for virtual circuit addressing are not universal — they are meaningful only for a particular link or router. For example, a switch might map a virtual circuit id for an incoming message to an outgoing link plus a new virtual circuit id to be used on the outgoing link. The addresses used in Internet addressing are universal — they mean the same thing everywhere. Internet addressing allows the route taken by a message to be chosen dynamically as the message flows through the network; virtual circuit addressing requires the route to be set up beforehand.

Problem 9 (12 points)

(A) (4 points)

The singleton sequence that contains the same external transition.

(B) (4 points)

The empty sequence.

(C) (4 points)

If `FetchAndStore` returns `nil` (so it changes `tl` from `nil` to `non-nil`), then the body of `Acquire` in the spec, else the empty sequence.

Problem 10 (10 points)

The intended answer to this problem is "no". `Acquire` can only fail to terminate by spinning forever in the loop at the end. Similarly, `Release` can only fail to terminate by spinning forever in the loop that waits for the next thread to link itself into the list. But the statement in `Acquire` that links a thread into the list comes after a bounded number of atomic actions in `Acquire`, so it is guaranteed to be executed. Thus, `Release` is guaranteed to terminate. Since threads are linked into the list in the order in which they execute `FetchAndStore`, a thread in `Acquire` will eventually reach the point where all the threads ahead of it on the list have acquired and then released the list, and will find its own flag true.

As a couple of people noticed, if threads use the module in ways other than was intended (but ways that are allowed by the specification given in the problem) — e.g., a thread could call `Acquire`, and then call `Acquire` again without first calling `Release` — strange things can happen. In particular, if a thread calls `Acquire` twice in a row without an intervening call of `Release`, the thread will spin forever in the loop at the end of `Acquire`. Similarly, if a thread calls `Release` when it is not the current holder of the mutex, things can go wrong. We intended to prevent this with the guard at the beginning of `Release`, but we forgot to have `Release` set `s(t).b` to `false` before returning — so after a thread `t` has acquired the mutex once, `s(t).b` will remain `true` even after it calls `Release`.

Problem 11 (10 points)

```
PROC FetchAndStore(t) -> TN =
   VAR tn, b: Bool := false |
   DO ~b =>
         tn := tl;
         b := CompareAndSwap(tn,t)
   OD;
   RET tn
```

Note: some people used `CompareAndSwap` in a guard; this is appealing, but is not legal Spec code, since guards are predicates, which are expressions, which cannot have side-effects.

Problem 12 (12 points)

(A) (4 points)
Yes. Suppose two threads call `Acquire` when the mutex is not held (so `tl=nil`), and they call `FetchAndStore` concurrently. If both read `tl` before either writes it, both will read `nil` from `tl`, and both will then return directly from `Acquire`.

(B) (4 points)
No. A thread in `Release` sets only one thread's flag, so only one spinning thread will return from `Acquire` as a result of a call of `Release`.

(C) (4 points)
Yes. Suppose two threads call `Acquire` when the mutex is held, and they call `FetchAndStore` concurrently. If both read `tl` before either writes it, both will read the same value for the previous tail of the list. Both will then attempt to link to that tail; the second one will overwrite the first. Whichever is the last to link to the previous tail before it tries to release the lock will then acquire the lock; the other will wait forever. Even worse, if the one that waits forever happens to be the one whose name ended up in `tl`, all other threads that call `Acquire` will be waiting for it, and they will also wait forever.

Problem 13 (8 points)
T1 calls `Acquire` and gets the lock, returning from `Acquire`.
T2 calls `Acquire`, which calls `FetchAndStore`, which returns T1.
T1 calls `Release`; `CompareAndSwap` returns false (because `tl` = T2, not T1).
T1 executes the body of the loop in `Release` one or more times.
T2 executes the statement `<< s(tn0).tn := t >>`.
T1 falls out of the loop in `Release`.