Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.826 Principles of Computer Systems

SOLUTIONS TO PROBLEM SET 2

Problem 1: Read-ahead Disk

There were some points of ambiguity in the problem statement. As long as your implementation did something reasonable, it was fine.

```
CLASS ReadAheadDisk EXPORT Byte, Data, DA, E, DBSize,
                           read, write, size, check, sync, Crash =

TYPE % Byte, Data, DA, DB, Block, E as in Disk

CONST N := 50

VAR udisk : Disk
    cache : DA -> DB := {}

APROC new(size: Int) -> ReadAheadDisk = <<
     self := StdNew(); udisk := udisk.new(size); RET self >>

PROC read(e) -> Data RAISES {notThere, error} =
     check(e); AddErrors();
     VAR data := Data{}, da := e.da, upTo := e.da + e.size;
         DO da < upTo =>
              IF cache!da => data + := cache(da); da + := 1
              [*] VAR i := RunNotInCache(da, upTo+N),
                  buffer := udisk.read(E{da, i}),
                  j := 0;
                  DO j < i =>
                    cache(da+j) := udisk.DToB(buffer(j))
                  OD;
              FI
         OD; RET data

PROC write(da, data) RAISES {notThere} =
     sync();
     udisk.write(da, data)

FUNC size() -> Int = RET udisk.size()

APROC sync() = << cache := {} >>

APROC check(e) RAISES {notThere} =
     << e.das.rng <= disk.dom => RET [*] RAISE notThere >>

PROC Crash() = CRASH; cache := {};
```

```
FUNC RunNotInCache(da, upTo: DA) -> I =
     RET {i | da + i <= upTo /\ ALL j :IN i.seq | ~cache!(da+j)}.max
```

b) Because writes are not buffered, the cache does not ever disagree with the disk:

```
% ABSTRACTION FUNCTION Disk.disk = udisk.disk
```

c) Initial state is that returned from `new`. The abstraction function clearly holds in this case.
We can simplify our proof by writing the invariant that the cache is a subset of the disk:

```
Inv(udisk:Disk, cache:DA -> DB) -> Bool =
    RET ALL da :IN cache.dom | cache(da) = udisk.dsk(da)
```

In the initial state, the cache is empty, so the invariant definitely holds. It is clear that `write`, `size`, `check`, `sync`, and `Crash` preserve the invariant, since at worst they clear the cache. We examine the effect of `read`. The cache is only mutated in the else-branch of the `IF` statement, where we have

```
buffer := udisk.read(E{da, i})
```

and

```
cache(da+j) := udisk.DToB(buffer(j))
```

Now consider `buffer(j)`. This gets read from `udisk.dsk(da+j)`, and is copied to `cache(da+j)`, proving that the invariant is preserved by `read`.

We proceed by a case analysis on `Disk` actions: `read`, `write`, `size`, `sync` (optional), `check`, `Crash`. Clearly, only the first two actions are interesting.

- `ReadAheadDisk.write` simulates `Disk.write`: We need to show only that `ReadAheadDisk.udisk.dsk` equals `Disk.dsk`, and that's clear because the read-ahead disk simply calls `udisk.write`.
- `ReadAheadDisk.read` simulates `Disk.read`: Note that the underlying abstract state is not changed by this action. The concrete state may change, but the invariant implies that the concrete state still corresponds to the abstract state. The return values are identical because `read` always returns data from the cache, which is guaranteed by the invariant to equal the data on the disk, which is what gets returned from `Disk.read`.

  There is one small technicality. If you're trying to prefetch data from the disk and it raises an error in the part being prefetched, your `read` might raise an error where the original `Disk` would have succeeded. We ignore this technicality.

Problem 2

a)

```
FUNC EToDAs(e) -> % as in Disk

module RAID4Disk[N] EXPORT Byte, Data, DA, E, DBSize, read, write, size, check, Crash =

TYPE E = {da, size : Nat}
         WITH {das:=EToDAs, dsks:=EToDsks, "IN":=(\ e, da | da IN e.das)}
     LB = SEQ Byte SUCHTHAT (\ db | db.size = DBSize * (N-1))

VAR  diskArray   : Int -> Dsk
```

```
% Some auxiliary functions
FUNC rdisk_noerr : LB = RET
    (\ i | + : LB { j :IN 1 .. N-1 | | diskArray(j)(i) })
% unless there's Null, where we return the XOR...
FUNC rdisk_recalculate_b : DB = RET
    (\ bad | (\ i | XOR : DB { j :IN (1..bad) + (bad+1..N) | |
                                            diskArray(j)(i)}))
FUNC rdisk_fix_err : LB = RET
    (\ i | + : LB { j :IN 1 .. N-1 | |
                          diskArray(j)(i) # Null => diskArray(j)(i)
                                          [*] rdisk_recalculate_b(j)(i) } )
FUNC rdisk : LB = RET (\i : DA | VAR r := rdisk_noerr(i) |
                      r.count(Null) = 0 => r
                      [*] r.count(Null) > 1 => Null
                      [*] rdisk_fix_err(i))

APROC new(size: Int) -> RAID4Disk = <<
       VAR i := 1 | DO i < N =>
           VAR dsk | dsk.dom = (size/(N-1)).seq.rng,
           diskArray(i) := dsk; i := i + 1 OD;
       VAR j := 0 | DO j < diskArray(N).size =>
           diskArray(N)(j) := XOR {k :IN 1 .. N-1 | | diskArray(k)(j)} OD;
       self := StdNew();
       RET self >>

PROC read(e) -> Data RAISES {notThere, error} = <<
       check(e); AddErrors();
       VAR dbs := e.das * rdisk |
           IF Null IN dbs => RAISE error [*] RET BToD(dbs) FI >>

PROC write(da, data) RAISES {notThere} =
       VAR blocks := DtoB(data), i := 0 |
           check(E{da, blocks.size});
           DO blocks!i => WriteBlock(da + i, blocks(i)); i + := 1 OD

PROC size() -> Int = RET diskArray(0).dom.size

APROC AddErrors() =
       << DO RET [] VAR da :IN disk.dom | disk(da) := nil OD >>

APROC WriteBlock(da, db) = <<
       VAR i := 1 | DO i <= N-1 =>
           diskArray(i)(da) := db.seq(i*DB, (i+1)*DB-1); i + := 1 OD;
       diskArray(N)(da) :=
           XOR { j :IN 1 .. N-1 | | db.seq(j*DB, (j+1)*DB-1)} >>

% various other uninteresting methods
```

b)

```
FUNC AF() -> Disk.disk = rdisk
```

If there are no errors, the abstraction function concatenates the $N-1$ sectors on the data disks to a sector of the logical disk; otherwise, it reconstructs one damaged sector by `XOR`ing all other sectors, and then concatenates the $N-1$ resulting sectors (in order, of course). Note that `read` essentially just uses the abstraction function to do its read.

c) We use an invariant that says that if there are no errors, then the XOR disk actually stores the XOR of the other $N-1$ disks:

```
Inv() = (ALL i :IN diskArray(N).dom |
          (EXISTS j :IN 1..N | diskArray(j)(i) = nil) \/
           diskArray(N)(i) = XOR : \{ j :IN 1..N-1 | | diskArray(j)(i))
```

Notice that the invariant holds on the initial state by construction. The `read` operation preserves the invariant if no errors are added, because the state is not mutated. If errors do get added, then for the logical sectors in which errors are added, the invariant holds by the first disjunct. Finally, `write` preserves the invariant by explicitly ensuring it with an assignment to the `XOR` disk.

To show that `RAID4Disk` implements `Disk`, note first that the initial state of our `RAID4Disk` corresponds to an initial state of `Disk`: the result of `RAID4Disk.new` is exactly the same as the result of `Disk.new`, because analogous code is used (and the invariant holds, of course).

We now argue that `read` and `write` implement their spec; the other procedures are uninteresting. Since `RAID4Disk.read` makes no changes in either abstract or concrete state, it remains only to show that it returns the same result as `Disk.read`. This follows from the fact that it uses exactly the abstraction function to read the abstract disk contents. It is important to also realize that any `read` call that returns an error can be replicated by adding an approriate error in `Disk.AddErrors`.

Finally, by considering the disk state after a call to `WriteBlock`, we see that applying the abstraction function `rdisk` to the post-write disk state gives the abstract state: concatenating

```
diskArray(i)(da) := db.seq(i*DB, (i+1)*DB-1);
```

makes the abstract state,

```
+ :  LB { j :IN 1 ..  N-1 | | diskArray(j)(i) }),
```

correspond to what was written to disk, so that the abstraction function correctly relates abstract and concrete in the post-state.