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

PROBLEM SET 2

Issued: Tuesday, February 12, 2002 **Due: Thursday, February 21, 2002 (updated)**

The following problems explore implementations and use of file system interfaces.

There are three problems in this problem set; please turn in each problem on a separate sheet of paper. Also give the amount of time you spend on each problem.

Problem 1. Encrypted File System

In this problem your task is to implement the `CipherFile` module. The `CipherFile` module is a filter for the `File` module at page 6 of Handout 7: it implements the interface of `File` and uses another instance of `File` internally. The module is parametrized by a pair of a coding and a decoding function that are mutually inverse functions:

```
MODULE CypherFile[
                  [code   : Byte -> Byte,
                   decode : Byte -> Byte]
                   SUCHTHAT \ [cd,dcd] |
                                cd * dcd = id /\
                                dcd * cd = id]
```

Here `id` denotes the identity function

```
id = \ x | x
```

A file is represented as a sequence of *encoded* bytes:

```
TYPE File = SEQ Byte
     D = PN -> F
VAR d := D{}
```

Ignore the possibility of crashes in this problem.

a) Write the abstraction function from `CipherFile.d` to `File.d`.
b) Implement operations `Read` and `WriteAtomic` in `CypherFile`.
c) Using the abstraction function in *a*), show that `CypherFile` implements `File`.
d) Is it true in your implementation that, conversely, `File` implements `CypherFile`?

Problem 2. Run-length Encoding File System

In this problem, your task is to design a file system that stores the file contents in compressed form. You should formulate this file system as an implementation of the following `MicroFile` interface, which is a simplification of the interface at page 6 of the Handout 7. The `MicroFile` interface ignores the problem of crashes.

```
MODULE MicroFile EXPORT PN, Byte, Data, X, F =
TYPE PN = String     % Path Name
            WITH {read:= Read, append:= Append}
     I = Int
     Byte = IN 0 .. 255
     Data = SEQ Byte
```

```
     X = Nat          % byte-in-file indeX
     F = Data         % File
     D = PN -> F      % Directory
VAR  d := D{}         % undefined everywhere
FUNC Read(pn: PN, x: Nat, i: Nat) -> Data =
  RET d(pn).seg(x,i)
APROC Append(pn: PN, b: Byte) = <<
  d(pn) := d(pn) + { b } >>
```

Compression in this file system is achieved by storing every sequence of n adjacent bytes b as a pair $\langle b, n\rangle$. The file content is then represented as a sequence of such pairs and has type `CompData`.

```
MODULE CompressFile
TYPE CompData = SEQ Pair
     Byte = MicroFile.Byte
     Pair = (Byte, PosInt)
     PosInt = Int SUCHTHAT (\i | i > 0)
     D = PN -> CompData
VAR  d := D{}
```

For example, the sequence of bytes `'Hello'` is represented as a list

```
[('H',1), ('e',1), ('l',2), ('o',1)]
```

(Here we have written a character sequence in single quotes as an abbreviation for the corresponding byte or a sequence of bytes.)

a) Write the abstraction function mapping `CompressFile.d` to `MicroFile.d`.
b) The compressed content of every file should be as short as possible. Find a simple sufficient condition on the values of `CompData` values that guarantees the shortest encoding. Use `SUCHTHAT` construct to write this condition as a `SPEC` invariant associated with the `CompData` type.
c) Write the implementation of the `Read` and `Append` operations in the `CompressFile` module.
d) Consider the set of states that can be generated by the `Read` and `Append` operations on an existing file in your implementation. Are there multiple states in `CompressFile` that the abstraction function maps to the same state in `File`?
e) Use the abstraction function to show that your `CompressFile` implements the `MicroFile` module.

Problem 3. Cache Replacement Policies

The procedure `BufferedDisk.MakeCacheSpace` in Handout 7 is not defined. Your goal in this problem is to write two different implementions of this procedure using two different cache replacement policies: LRU and LRUCF.

a) LRU (Least Recently Used) policy makes space in the cache by first replacing the block that was accessed least recently. Write the implementation of `BufferedDisk.MakeCacheSpace` using LRU policy.
b) LRUCF (Least Recently Used, Clean First) policy makes space in the cache by first replacing blocks that are clean (not modified compared to their original content). Among the clean blocks, it first replaces those that were least recently used, using LRU policy. If there are no clean blocks, it choses among the dirty blocks using LRU policy. Write the implementation of `BufferedDisk.MakeCacheSpace` using LRUCF policy.

Try to minimize the number of changes you make to the `BufferedDisk` implementation from Handout 7. For the changes you make, argue informally that the same abstraction function can still be used to show that `BufferedDisk` implements `BDisk` module from Handout 7.