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

PROBLEM SET 1

Assigned: February 8, 2006 Due: February 15, 2006

Problem 0: Factoring (10 points)

Integer factorization is an intriguing problem: we know how to efficiently determine whether a number is prime or composite, but we cannot quickly find the prime factors of a composite number. Many deployed encryption schemes rely on factoring remaining computationally intractable.

a) (5 points) Write a `Spec` specification for a procedure which takes an integer $n > 0$ and returns a nontrivial integer factorization (p, q), such that $(pq = n) \wedge (p, q > 1)$, or fails.

b) (5 points) Write a simple, deterministic `Spec` implementation for this procedure, using trial division — that is, try all integers from 2 up to $\frac{n}{2}$. Throw an exception if n is not factorable.

Problem 1: Priority Queues (30 points)

A priority queue is a data structure which has the following basic operations: 1) insert, 2) find-minimum, and 3) delete-minimum. Here is a specification for a priority queue:

```
MODULE PriorityQueue[T WITH {"<="}]
                   EXPORT Insert, FindMin, DeleteMin
    VAR q : T -> Int := (\ t | 0 )

    APROC Insert(n : T) = << q(n) := q(n) + 1 >>

    FUNC isEmpty() -> Bool = (ALL t | q(t) = 0) => true [*] false

    APROC FindMin() -> T RAISES {Empty} =
                    IF isEmpty() => RAISE Empty
                                [*] RET {t | q(t) > 0 | t}.min

    APROC DeleteMin() -> T RAISES {Empty} =
                    << IF isEmpty() => RAISE Empty
                                [*] VAR m = FindMin() | q(m) := q(m) - 1 >>
END PriorityQueue
```

a) (10 points) Provide a `Spec` priority queue implementation (Cormen, Leiserson and Rivest have a good priority queue implementation based on a heap). All operations should take at most $O(\log n)$ time.

b) (20 points) State (in `Spec`) the invariants maintained in your solution, as well as an abstraction function from the priority queue to the multiset used in the above specification.

Problem 2: N-way set associative cache (40 points)

In this problem, we provide yet another implementation of the `Memory` module, this time for an *N-way set associative cache*.

An N-way set associative cache is divided into B buckets of N lines each (for a total of $N \cdot B$ lines). We assume the existence of a simple hash function hf mapping the set of addresses to $[0 \ldots B-1]$. Address a

is mapped to block $hf(a)$ and may be stored in any of the N locations within that block. Note that the write-back cache example of Handout 5 resembles an N-way set associative cache with $B = 1$, while the hash table example of Handout 5 resembles an N-way set associative cache with $N = 1$.

```
MODULE SetAssociative[A WITH {hf: A->Int}, V, N, B]
     EXPORT Read, Write, Reset, Swap =

TYPE M = A -> V                % main memory
...
```

a) (10 points) Provide a `Spec` implementation of `SetAssociative`. As suggested above, the implementation should use a main memory variable of type `A -> V` in conjunction with the cache.

b) (10 points) Write down the abstraction function `AF` mapping its state to the state of the `Memory` module.

c) (20 points) Give a formal proof that `AF` satisfies the definition of an abstraction function.

Problem 3: Logging File System (20 points)

Tired of losing old versions of files, you decide to implement a file system that maintains all of the versions of every file that has ever been written. The basic idea is that the file system stores, in addition to the file name, a version number for each version of the file. Here is the specification; note that each file is a string and that `Write`, `Read`, and `ReadLatest` operate on complete files.

```
MODULE FileVersion EXPORT Write, Read, ReadLatest =
  VAR fs : (String,Int) -> String := {},
      vs : String -> Int := {}

  APROC Write(n:String, f:String) = <<
    ~(n :IN vs.dom) => vs(n) := 0;
    fs(n,vs(n)) := f; vs(n) := vs(n)+1
  >>

  APROC Read(n:String, i:Int) -> String RAISES {NoFile} =
    << (n,i) :IN fs.dom => RET fs(n,i) [*] RAISE NoFile >>

  APROC ReadLatest(n:String) -> String RAISES {NoFile} =
    << ~(n :IN vs.dom) => RAISE NoFile [*] RET fs(n,vs(n)) >>
END FileVersion
```

You are implementing your file system on top of a write-once, append-only storage medium. The storage medium also allows you to read arbitrary records that you have stored in this medium.

```
MODULE WriteOnceMemory[T] EXPORT Append, Fetch =
  VAR store : SEQ T

  APROC Append(n:T) -> Int = << VAR l = store.size;
                               store := store + n; RET l >>

  FUNC Fetch(i:Int) RAISES {NotThere} -> i > store.size => RAISE NotThere
                                        [*] T = store(i)
END WriteOnceMemory
```

Your assignment is to implement the versioning file system on top of this medium.