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

PROBLEM SET 2 SOLUTIONS

Problem 1. Encrypted File System

a)

```
ABSTRACTION FUNCTION File.d = \ pn |
   CipherFile.d(pn) * decode
```

b)

```
FUNC CipherFile.Read(pn, x, i) =
     MicroFile.Read(pn, x, i) * decode
APROC CipherFile.WriteAtomic(pn, x, i) =
     MicroFile.WriteAtomic(pn, x, data * code)
```

c) We present a detailed proof to illustrate the technique. We use the sufficient condition for inclusion of sets of traces given in Handout 6, page 7. Note that we are dealing with two instances of the `File` in this proof: one used to implement `CipherFile` and one against which we are comparing the implementation.

Base Case. We need to show that the initial state of `CipherFile` is mapped to the initial state of `File`. The initial state of `CipherFile` is everywhere undefined map `D`, and the abstraction function maps `D` to everywhere undefined map `D`, which is the initial state of `File`. So the base case holds.

Induction Step. Let $cs_1 \xrightarrow{\tau} cs_2$ be a transition in `CipherFile` where cs_1 and cs_2 are states of `CipherFile` and τ is either `WriteAtomic(pn,x,data)` or `Read(pn,x,i)->data` for some values of `pn`,`x`, and `data`. (Here cs stands for "CipherFile state".)

Let `c2f` denote the abstraction function from *a)* part. We need to show that there exists a transition $\mathsf{c2f}(cs_1) \xrightarrow{\tau} \mathsf{c2f}(cs_2)$ in `File`.

```
c2f(cs1) ---> s2
/|\             c2f(cs2)
 |                /|\
 |                 |
 |                 |
cs1 ------------> cs2
```

Consider first the case of `WriteAtomic(pn,x,data)`. Let s_2 be the result of applying `File.WriteAtomic(pn,x,data)` to $\mathsf{c2f}(cs_1)$. We want to show that $s_2 = \mathsf{c2f}(cs_2)$.

All files `pn'` $\neq$ `pn` are unmodified by the `WriteAtomic` transition in both `CipherFile` and `File`. Therefore

$$\begin{aligned} s_2(\texttt{pn'}) &= \mathsf{c2f}(cs_1)(\texttt{pn'}) \\ &= cs_1(\texttt{pn'}) * \texttt{decode} \\ &= cs_2(\texttt{pn'}) * \texttt{decode} \\ &= \mathsf{c2f}(cs_2)(\texttt{pn'}) \end{aligned}$$

Next we need to show that

$$s_2(\texttt{pn}) = \mathsf{c2f}(cs_2)(\texttt{pn})$$

Given the definitions of `WriteAtomic` in our case, this reduces to:

$$\texttt{NewFile}(cs_1(\texttt{pn}) * \texttt{decode}, x, \texttt{data}) = \texttt{NewFile}(cs_1(\texttt{pn}), x, \texttt{data} * \texttt{code}) * \texttt{decode} \tag{1}$$

This means that the sequences need to be equal at every index i. Consider the structure of `NewFile`. Assuming that the write does not start after the end of file, we have that `NewFile` either retains the original value or replaces it with the value of argument. If the original value is preserved, then the i-th element in the sequence on both sides is $\texttt{decode}(cs_1(\texttt{pn})(i))$. If the value is overwritten, then the i-th element in the sequence on left hand side is

$$\texttt{decode}(\texttt{code}(\texttt{data}(i-x)))$$

and the element on the right hand side is `data(i-x)`, which is equal because `decode` and `code` are inverse functions. Here x is the length of the `pn` files. (`CypherFile` and `File` have equal lengths, by the abstraction function.)

Note. We used the assumption that there are no writes that start past the end of the file. Without that assumption, the left hand side would evaluate to 0 and the right hand side to `code(0)` so the abstraction function would not work.

Next we consider the case of `Read(pn,x,i)->data` in `CipherFile`. Because `Read` is a function, it does not modify the state. This means that $cs_2 = cs_1$. All we need to show is that we can apply a transition `Read(pn,x,i)->data` in `File` i.e. that the results of two function calls are the same. The transition in `File` does not modify the state either so the two states resulting from transition will be clearly related by c2f.

To show that the two results of `Read(pn,x,i)` are the same, we use the definition of `Read` operation in `CipherFile` and `File`:

$$\begin{aligned}
\texttt{CipherFile.Read(pn,x,i)} &= cs_1(\texttt{pn})\texttt{.seg(x,i)} * \texttt{decode} \\
&= (cs_1(\texttt{pn}) * \texttt{decode})\texttt{.seg(x,i)} \\
&= \mathsf{c2f}(cs_1)\texttt{.seg(x,i)} \\
&= \texttt{File.Read(pn,x,i)}
\end{aligned}$$

d) Yes, `File` implements `CypherFile` as far as `Read` and `WriteAtomic` operations are considered. It is sufficient to use the following abstraction function.

```
ABSTRACTION FUNCTION CypherFile.d = \ pn |
   File.d(pn) * code
```

The proof that implementation meets the specification is analogous to the *c*) part.

Problem 2. Run-length Encoding File System

a)

```
FUNC expand(p: Pair) -> SEQ Byte =
  VAR b : Byte, n: Int | (b,n) = p =>
  VAR s : SEQ Byte | s.size = n /\ ALL i: IN 0..n-1 | s(i)=b =>
  RET s

FUNC expandAll(data: SEQ Byte) -> SEQ Byte =
  + : (data * expand)

ABSTRACTION FUNCTION MicroFile.d = CompressFile.d * expandAll
```

b)

```
TYPE CompData = SEQ Pair SUCHTHAT (\s |
                ALL i: 0 .. s.size-2 | ~(fst(s(i))=fst(s(i+1)))
```

c) The following implementation avoids expanding part of the file that is skipped while reading.

```
FUNC DataRead(cd: CompData, i: Nat) -> Data =
  VAR b: Byte, n: Int | (b,n) = cd.head =>
  IF (i >= n) => RET expand(b,n) + DataRead(cd.tail, i-n)
  [*] RET expand(b,i)

FUNC DataSeek(cd: CompData, x: Nat, i: Nat) -> Data =
  VAR b: Byte, n: Int | (b,n) = cd.head =>
  IF (x >= n) => RET DataSeek(cd.tail, x-n, i)
  [*] RET DataRead({(b,n-x)} + cd.tail, i-x)
  FI

FUNC Read(pn: PN, x: Nat, i: Nat) -> Data =
  DataSeek(d(pn),x,i)

APROC fix(cd: CompData) -> CompData = <<
      VAR cdPre: CompData,
          b: Byte, n1: PosInt, n2: PosInt |
      cd = cdPre + { <0, <b,n1>>, <1, <b,n2>> } =>
      RET cdPre + { <0, <b,n1+n2>> }
  [*] RET cd >>

APROC Append(pn: PN, b: Byte) = <<
  d(pn) := fix(d(pn) + { <b,1> }) >>
```

d) No. Provided that the invariant in *b*) holds, the abstraction function is a bijection between the two spaces. This means that there is exactly one `CompressFile.d` value for each `MicroFile.d` value.

e) The base case is easy because both systems start with en empty file. The inductive case needs to show the correctness of `Append` and `Read` implementations.

Note. As in the formulation of the problem set formulation we use notation `x` to denote a list of length 1. This is just a shorthand that means `<0,x>` namely a seqence (function) that maps 0 to `x` and is undefined everywhere else.

`Append` specification adds a byte to the end of the file. `Append` implementation first adds a pair of byte and number 1 to the end of the file, and then calls `fix`. The correctness of the implementation follows from:

```
expandAll(cd + { <b,1> }) = expandAll(cd) + { b }
```

and

```
expandAll(fix(cd)) = expandAll(cd)
```

The last equation holds because

```
expandAll(cdPre + { <0, <b,n1>>, <1, <b,n2>> })
= expandAll(cdPre) + expandAll({<b,n1>}) + expandAll({<b,n2>})
= expandAll(cdPre) + expandAll({<b,n1+n2>})
= expandAll(cdPre + {<b,n1+n2>})
```

To show the correctness of `Read`, we show that

```
DataSeek(cd,x,i) = expandAll(cd).seg(x,i)
DataRead(cd,i) = expandAll(cd).seg(0,i)
```

The proof is by induction which corresponds to the recursive definitions of `DataSeek` and `DataRead`. (This induction is well founded because the length of `data` decreases in each recursive call.)

Correctness of `DataRead`**.** The recursion terminates always on a list `cd` of length at least one.

If $i < n$ then

```
DataRead({<b,n>}+cd,i) = expand(b,i) = expand(b,n).seg(0,i)
= expandAll({<b,n>}+cd).seg(0,i)
```

If $i \geq n$ then

```
DataRead({<b,n>}+cd,i) = expand(b,n) + DataRead(cd,i-n)
= expand(b,n) + expandAll(cd).seg(i-n)
= expandAll({<b,n>} + cd).seg(0,i)
```

Correctness of `DataSeek`**.** The recursion again terminates always on a list `cd` of length at least one.

If $x < n$ then

```
DataSeek({<b,n>}+cd,x,i) = DataRead({<b,n-x>}+cd, i-x)
= expandAll({<b,n-x>}+cd).seg(0,i-x)
= expandAll({<b,x>}+{<b,n-x>}+cd).seg(x,i)
= expandAll({<b,n>}+cd).seg(x,i)
```

If $x \geq n$ then

```
DataSeek({<b,n>}+cd,x,i) = DataSeek(cd,x-n,i)
= expandAll(cd).seg(x-n,i)
= expandAll({<b,n>}+cd).seg(x,i)
```

Problem 3. Cache Replacement Policies

We first observe that the `queue` variable is not sufficient for maintaining the information on which block was most recently used, because it only keeps track of the dirty blocks. (It's purpose is to ensure that the writes are performed in order.)

We therefore introduce another list `usage` which contains all valid blocks of the cache from the most recently used to the least recently used. We define internal procedures `cacheRead` and `cacheUpdate` used to read and write the cache. We then make these operations maintain the information on which block was used most recently in the `usage` list.

```
VAR usage: SEQ DA;
INVARIANT Subset(usage.rng, cache.dom)

FUNC removeElem(da: DA, u: SEQ DA) =
  IF  u.size=0 => {}
  [*] u.head=da => u.tail
  [*] { u.head } + removeElem(da, u.tail)
  FI

APROC use(da) = << usage := { da } + removeElem(da,usage); >>

APROC cacheRead(da) -> DB = <<
  use(da);
  RET cache(da) >>

APROC cacheUpdate(da, db) = <<
  use(da);
  cache(da) := db; >>
```

We replace all reads updates to cache in `read` and `write` operations in the `BufferedDisk` implementation with the calls to `cacheRead` and `cacheUpdate`. We can now define `MakeCacheSpace` that implements the desired policies.

a) LRU

```
FUNC nowFree() -> Int = cacheSize - cache.dom.size

APROC evict(da: DA) = <<
  DO VAR i: Int | queue(i)=da => FlushQueue(i) OD;
  cache(da) := undefined >>

APROC evictLRU() = << evict(usage.last); usage := usage.reml >>

PROC MakeCacheSpace(i: Int) -> Int =
  VAR canGet: Int := min(cacheSize, i) |
  DO << (nowFree() < canGet) => evictLRU() >> OD
  RET canGet
```

b) LRUCF

```
FUNC removeith(i: Int, s: SEQ DA) =
  s.seg(0,i-1) + s.seg(i+1,s.size-1)

APROC evictLRUCF() = <<
  VAR usageClean: SEQ DA :=
    { j :IN usage.dom | clean(usage(j)) } |
  IF usageClean.size=0 => evict(usage.last); usage := usage.reml
  [*] evict(usage(usageClean.last));
      usage := removeith(usageClean.last, usage)
  FI >>

PROC MakeCacheSpace(i: Int) -> Int =
  VAR canGet: Int := min(cacheSize, i) |
  DO << (nowFree() < canGet) => evictLRUCF() >> OD
  RET canGet
```

The correctness of the implementation follows from the fact that `cacheRead` and `cacheUpdate` peform the same actions on the cache and return the same results as the read and update operations on `cache`. `MakeCacheSpace` evicts blocks from the cache, but does so only through `evict` procedure, which in turn uses `FlushQueue`. This ensures that block is written back before being evicted, and it ensures that the order of writes is preserved because it is always an entire prefix of the `queue` that is written back.