Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 35 November 3, 1993

Solutions to Problem Set #3

The maximum score for this problem set was 75 points.[1]

Problem 1 (10 points)

In this question you are to give a precise definition of the semantics for an alternative choice operator on commands. The "or" operator on commands is sometimes called "demonic choice", because if either of the component commands loops indefinitely, the composite command can also loop.

So, for example, the command "`DO SKIP OD [] x:=3`*" can loop. (It can also set* `x` *to* `3` *and terminate.) An alternative operator, which we will denote by the symbol "*`<>`*", is sometimes called "angelic choice" because it is guaranteed to terminate if either of its component commands terminates. Thus, for example, the command "*`DO SKIP OD <> x:=3`*" has only one possible outcome, which involves setting* `x` *to* `3` *and terminating.*

In the style of Handout 14, give a precise definition of the semantics of `<>`*.*

We need a definition for `MC(c1<>c2)(s,o)` in terms of `MC(c1)` and `MC(c2)`. The cases are as follows:

- If both `c1` and `c2` loop forever, then `c1<>c2` loops forever.
- If either `c1` or `c2` terminates, then `c1<>c2` terminates.

There is still some nondeterminism possible in the case that either `c1` or `c2` is nondeterministic. For example, by the definition above,

```
BEGIN DO SKIP OD [] x:=3 END  <>  BEGIN DO SKIP OD [] x:=3 END
```

can still possibly loop forever, or can terminate with `x=3`.

Since, for a given starting state `s`, there is only one final state `s{"$x"->loopX}` which denotes an infinite loop, the following predicate suffices, corresponding to the two cases given above:

```
( MC(c1)(s,o) /\ MC(c2)(s,o) ) \/
( ( MC(c1)(s,o) \/ MC(c2)(s,o) ) /\ o("$x") # loopX )
```

[1] No, these points are not weighted the same as those for PS #1.

Problem 2 (15 points)

Consider the following specification of a storage manager:

```
MODULE StrgMgr [T] =

TYPE S = SEQ[T]            % The allocated storage.
     N = Int               % Names for allocated entities.
     Freed = SET[N]        % Set of freed names.

VAR s := S{}
    freed := Freed{}

APROC Malloc (t) -> N =    % Allocate and initialize a new T.
  << VAR n := s.size |
     s := s ++ t;
     RET n >>

APROC Free (n) =           % Free the storage named by n.
  << n IN freed \/ n < 0 \/ n>=s.size => HAVOC
    [*] freed := freed ++ n >>

APROC Write (n,t) =        % Write a new value at location n.
  << n IN freed \/ n < 0 \/ n>=s.size => HAVOC
    [*] s(n) := t >>

APROC Read (n) -> T =      % Read the value stored at location n.
  << n IN freed \/ n < 0 \/ n>=s.size => HAVOC
    [*] RET s(n) >>

END StrgMgr
```

Now consider the following implementation, which attempts to minimize the amount of storage in use:

```
MODULE StrgMgrImpl [T] =

TYPE S = SEQ[T]            % The allocated storage.
     N = Int               % Names for allocated entities.
     Free = SET[N]         % Indexes of s available for reuse.
     OMap = N -> N         % Map from external to internal names.

VAR s := S{}
    omap := OMap{}
    free := Free{}
```

```
    nextN: N := 0           % Next external name to allocate.

APROC Malloc (t) -> N =     % Allocate and initialize a new T.
  << free.size = 0 =>
           s := s ++ t;
           omap(nextN) := s.size-1;
           nextN := nextN+1;
           RET nextN-1
   [*] VAR n | n IN free =>
           free := free - n;
           s(n) := t;
           omap(nextN) := n;
           nextN := nextN+1;
           RET nextN-1 >>

APROC Free (n) =            % Free the storage named by n.
  << free := free ++ omap(n) >>

APROC Write (n,t) =         % Write a new value at location n.
  << s(omap(n)) := t >>

APROC Read (n) -> T =       % Read the value stored at location n.
  << RET s(omap(n)) >>

END StrgMgrImpl
```

Use history variables to define an abstraction function for the implementation, and sketch a proof of correctness using your abstraction function and any invariants that you need. Note: your abstraction function will have to define a mapping on `N`*'s (*i.e.*, the names used for allocated entities), since the implementation recycles them.*

The easiest (though not always cleanest) way to use history variables is to simply add the entire state of the specification to the state of the implementation as a set of history variables. The abstracton function then becomes trivial, leaving only the task of finding a suitable rep. invariant.

Following this scheme, we get

```
MODULE StrgMgrImplH [T] =

TYPE ...                    % As before

VAR ...                     % As before
    hs := S{}               % History variables
    hfreed := Free{}
```

```
APROC Malloc (t) -> N =
  << hs ++ t;
     ...                    % Rest as before
  >>

APROC Free (n) =
  << hfreed := hfreed ++ n;
     free := free ++ omap(n) >>

APROC Write (n,t) =
  << hs(n) := t;
     ... >>                 % Rest as before

APROC Read (n) -> T =
  ...                       % As before

END StrgMgrImplH
```

Note that we have to be careful to ensure that no procedure's outcome depends on the values of the history variables.

The abstraction function then is simply `AF(...,hs,hfreed) = (hs,hfreed)`.

Now comes the interesting part. To get a proof of correctness, we must have a rep. invariant that ensures that the (spec.) state represented by the history variables really does match the state of the implementation:

```
NextN = hs.size
/\ (ALL n,m | (hs!n ==> hs(n) = s(omap(n)) \/ n IN hfreed)
            /\ (omap!n ==> hs!n)
            /\ (n IN free /\ n = omap(m) ==> m IN hfreed))
```

What is important here is not really that you be able to determine the values of all implementation variables from the history variables (though that would be sufficient) but that you have enough information to do the correctness proof.

The proof of correctness would then, as usual, involve first proving that the rep. invariant was really a rep. invariant, and then proving that each transition between states `s` and `s'` of the implementation had a corresponding transition in the specification between states `AF(s)` and `AF(s')`.

As an example of a case from the proof of the rep. invariant, take a transition `(Malloc(t),n)` from a state `s` in which the rep. invariant holds to a new state `s'`. First, no matter which branch is taken, `s'.nextN = s.nextN+1` and `s'.hs = s.hs++t`, so s'.hs.size = s.hs.size+1, so the first line of the invariant is satisfied.

Next, assume `s.free.size = 0`. The only value of `n` for which either `hs(n)` or `s(omap(n))` changes is `s.nextN`, and both are set to `t`, so the invariant is preserved. On the other hand, if `n IN s.free`, then in addition to `hs(s.nextN)` and `s(omap(nextN))`, we may also be changing the value of `s(omap(m))` for some other `m`. However, the last line of the rep. invariant guarantees that any such `m` is in `hfreed`, so the rep. invariant is still satisfied in `s'`.

As an example from the proof of the abstraction function, consider a transition `(Read(n),t)` in the implementation between states `s` and `s'`. If `s.hs!n` then the rep. invariant implies that either `s.omap!n` or `n IN freed`. In the latter case, anything the implementation does matches the spec, since the spec in that case is `HAVOC`. In the former, `t=s.s(omap(n))=s.hs(n)`, so, with the given abstraction function, there is a transition `(Read(n),t)` from `AF(s)` to `AF(s')`

Problem 3 (10 points)

In the previous question, we would have liked to implement the storage manager without using the omap. What assumptions would we need to make about the clients of the storage manager module for such an implementation to be correct?

Any client that makes any assumptions at all about the numbers returned by `Malloc` will be able to detect a difference between the implementation and the specification. The easiest way to phrase the necessary restriction is to require that the only operations a client may employ on values returned by `Malloc` are `Free`, `Write`, `Read` and `:=` (*i.e.*, the values may be stored).

Problem 4 (40 points)

Recall the standard Unix representation of one file as it was explained in class.

(a) (15 points)

Write in Spec the state for this representation; decide any minor details that you don't remember or that weren't given in class any way you like. Give the abstraction function from this representation to the specification of the state of one file given in Handout 9.

The following is the state for the standard UNIX representation of one file.

```
MODULE UnixFile

Type

  Byte = File.Byte                % Int        0-225
  Data = File.F                   % SEQ Byte
  PN   = File.PN                  % String     Path Name
  Dir  = File.Dir                 % PN -> F    Directory
  DA   = Dsk.DA                   % Int        Disk Block Address,
                                  %            -1 is special non-address
  E    = Dsk.E                    % RECORD[at:DA, size:Int]
  Disk = Dsk.Disk                 % SEQ DB

  SDA  = SEQ DA                   % sequence of addresses
  Inode = RECORD [ size: Int,     % file data size
                   beg: SDA,      % direct blocks addresses
                   mid: DA,       % indirect block address
```

```
                    end: DA,       % doubly indirect block address
                    beyond: DA ]   % triply indirect block address
  F    = Inode                     % representation of a file is an inode

VAR
  dir    := Dir{}
  disk   := Disk{}
  MaxDirect                        % constant max. number of direct disk blocks
                                   % in an inode (eg. 10 direct disk blocks)
```

The abstraction function maps this inode representation of a UNIX file to a file given in Handout 9.

```
FUNC AF (f) -> File.F =
    VAR inode := f                 % F = Inode
        sda  := Flatten (disk, inode), % flatten the inode to a sequence
                                       % of addresses
        data  := Data{} |          % Data = File.F
        DO sda # SDA{} =>
            BEGIN
              sda.head # -1 =>     % skip any special non-address
                data := data + disk.ReadBlocks(E{at:=sda.head,size:=1})
            END;
            sda := sda.tail
        OD;
        % makes sure the correct size of file data is returned
        % for file size may not be divisible by data block size
        RET data.sub(0,inode.size-1) % Data = File.F
```

Auxiliary function `Flatten` puts the direct block addresses, flattened indirect block addresses, flattened doubly indirect block addresses, and flattened triply indirect block addresses together

```
FUNC Flatten(disk, inode) -> SDA =
  VAR sda, i: Int := 0 |
      DO i < MaxDirect =>
          BEGIN inode.beg[i] # -1 => sda := sda + inode.beg[i] END
      i = MaxDirect =>             % all direct block addresses are used
          sda := sda + FlattenInd (disk, inode.mid, 0)
                     + FlattenInd (disk, inode.end, 1)
                     + FlattenInd (disk, inode.beyond, 2);
      RET sda
```

Auxiliary function `FlattenInd` flattens the indirect block addresses recursively with a maximum bound of 3 level of recursion for triply indirect block addresses

```
FUNC FlattenInd(disk, da, level:Int) -> SDA =
  da = -1 =>
    RET -1                      % return special non-address if it is
  [*] VAR sda |
        sda := DataToSDA(disk.ReadBlocks(E{at:=da,size:=1});
        level = 0 => RET sda    % return addresses in an indirect block
        [*] BEGIN               % recursively invokes FlattenInd for
                                % doubly/triply indirect block addresses
              VAR sda' := SDA{} |
                DO sda # SDA{} =>
                    sda' := sda' + FlattenInd(disk,sda.head,level-1);
                    sda := sda.tail
                OD;
                RET sda'
            END;
```

Auxiliary function `DataToSDA` simply groups the bytes in the file data into a sequence of disk addresses

```
FUNC DataToSDA(Data) -> SDA = ...

...

END UnixFile
```

(b) (15 points)

Write in Spec the state for the log used in the log structured file system as it was explained in class, assuming that the standard Unix representation of (a) is used as the index structure. For this purpose treat the log as a `SEQ Byte` *rather than as a collection of disk pages. Give the abstraction function from the representation of a single file in your log to the standard Unix representation.*

The following is the state for the log used in the log file stuctured (LFS) file system using the standard UNIX representation of index structure. The log contains (1) inode maps, (2) inodes, and (3) blocks of data or log addresses.

```
MODULE LFS

Type

  Byte = File.Byte              % Int        0-255
  Data = File.Data              % SEQ Byte   Data
  PN   = File.PN                % String     Path Name
  Dir  = File.Dir               % PN -> F    Directory
```

```
  LA   = Int                      % log address, -1 is special non-address
  SLA  = SEQ LA                   % sequence of log addresses
  IN   = Int                      % inode number
  InodeMap = SEQ LA               % (1) inodemap maps IN to LA
  Inode= RECORD [ inum:IN,        % (2) inode block containing inode number
                  size:Int,       % file data size
                  beg:SLA,        % direct blocks addresses
                  mid:LA,         % indirect block address
                  end:LA,         % doubly indirect block address
                  beyond:LA ]     % triply indirect block address
  LB   = RECORD [ inum:IN,        % (3) log block of data with inode number
                  bnum:Int,       % block number in the file (data block)
                  content:Data ]  % content: a block of data
  F    = IN                       % rep. of a file is an inode number
  Log  = SEQ Byte                 % a log is a sequence of bytes

VAR
  dir    := Dir{}
  log    := Log{}
  MaxDirect                       % constant max. number of direct log blocks
                                  % in an inode (eg. 10 direct log blocks)
```

Additionally, the following types of variable are defined for the AF

```
Type
  LEC  = UNION [InodeMap,         % 3 types of log entry content, (1) inodemap
                Inode,            % (2) inode
                LB]               % (3) block of data or log addresses
  PLog = SEQ [la, lec]            % parsed log with each entry: address
                                  % and the corresponding content
```

According to the model of LFS taught by Professor Lampson in the class, the most updated inode in the log may not index to all the newly-written log blocks. It is only when an inodemap is written to the log, a new inode is also written to the log for each file which has been changed since last time an inodemap is written to the log. Therefore, we have to check if there is any newly-written log block after the last inodemap on the log and use it to construct the abstractly-correct inode for a file.

For simplicity, we identify LAs in LFS with DAs in UnixFile.

The abstraction function maps a log and a file representation in the log to a disk which contains UNIX inode representation of the corresponding file. In order for a LFS file to map to a UNIX file, three conditions must be satisfied: (1) the log block addresses in the flattened LFS file inode are equal to the disk block addresses in the UNIX inode (2) the log blocks contained in the LFS file inode have the same content as the disk blocks of the UNIX inode (3) the LFS file is of the same size as the UNIX file

```
FUNC AF(log, f) -> (UnixFile.Disk, UnixFile.F) =
  VAR plog := ParseLog(log),     % parse a SEQ Byte to a Plog
      in := f,                   % F = IN
      size, sla |                % size and updated log addresses of the file
    (size, sla) := UpdatedFlatten(log, plog, in);
    VAR disk : UnixFile.Disk, uinode : UnixFile.Inode |
        (   UnixFile.Flatten(disk, uinode) = sla              % cond (1)
         /\ (ALL la IN sla | disk[la] = ReadLog(log, la))      % cond (2)
         /\ uinode.size = size ) =>                            % cond (3)
        RET (disk, uinode)       % UnixFile.F = UnixFile.Inode
```

`ParseLog` parses the sequence of bytes in the log to a corresponding sequence of log entries which can be an inodemap, an inode, or a data log block.

```
PROC ParseLog(log) -> PLog =
  VAR plog,                       % find a plog that when encoded is the log
      contents : SEQ Log          % contents has contents of all plog entries
          := plog{x | EncodeLEC(x.lec)} |
        (   FileSystemImpl.Cat(contents) = log % concat of plog contents = log
         /\ (ALL i | plog!i ==>   % plog(i).la is size of earlier stuff
              plog[i].la = FileSystemImpl.Cat(contents.sub(0,i-1)).size) ) =>
        RET plog
```

`ReadLog` reads the content of a log entry. The la must be the start of an entry.

```
PROC ReadLog(log, la) -> LEC =
  VAR lec, log' | log = EncodeLEC(lec) + log' => RET lec
```

Here is the encoding function that maps LECs to bytes

```
FUNC EncodeLEC(lec) -> SEQ Byte = ...
```

`UpdatedFlatten` gets the last-written inode of the file and examines the portion of the parsed log after the last-written inodemap in order to construct the most updated abstract flattened inode of the file with addresses to the newly-written log blocks.

```
PROC UpdatedFlatten(log, in) -> (Int, SLA) =
  VAR plog0 := ParseLog(log),
      plog := plog0.sub(                  % the part of plog after
         FindInodeMap(plog), plog.size); % the last inodemap
      inodemap := plog[0],               % the last inodemap in plog
      inode := ReadLog(log,inodemap[in]),% get the inode shown in inodemap
      sla := Flatten(log, inode) |       % get the block addresses from inode
      % construct most updated seq of blocks by examining the plog'
    DO plog # {} => VAR la, lec |
```

```
            (la, lec) := plog.head;
            plog := plog.tail;
            lec IS LB /\ lec.inum = in =>     % a newly-written block is found
                sla[lec.bnum] := la           % update the log address in inode
            [*] SKIP
        OD
        RET (inode.size, sla)
```

`FindInodeMap` finds the last inodemap in the parsed log.

```
FUNC FindInodeMap(plog) -> Int = RET {i | plog[i].lec IS InodeMap}.max
```

Auxiliary function `Flatten` puts the direct blocks addresses, flattened indirect block addresses, flattened doubly indirect block addresses, and flattened triply indirect block addresses together. It is the same as `UnixFile.Flatten` except it deals with `Log` and `LA`, instead of `UnixFile.Disk` and `UnixFile.DA`, and in `FlattenInd`, `DataToSLA` is called instead of `DataToSDA` and `ReadLog(log, la)` is used instead of `disk.ReadBlocks(Eat:=at,size:=1)`.

```
FUNC Flatten(log, inode) -> SLA = ...

...

END LFS
```

(c) (10 points)

Add to the state you give in (b) the volatile state needed for a practical implementation as described in class (that is, a cache of all the index blocks whose abstract state differs from the state recorded on the disk). Give the invariant satisfied by the combined state. Write in Spec a description of the algorithm for reconstructing the volatile state from the log after a crash. You don't have to worry about the representation of Free, just about the indexing information.

The following is the state for the LFS with cache.

```
MODULE CacheLFS

Type

  Byte     = File.Byte          % Int         0-255
  Data     = File.Data          % SEQ Byte    Data
  PN       = File.PN            % String      Path Name
  Dir      = File.Dir           % PN -> F     Directory
  LA       = LFS.LA             % Int         Log Address
  SLA      = LFS.SLA            % SEQ LA      Sequence of LA
  IN       = LFS.IN             % Int         Inode Number
  InodeMap = LFS.InodeMap       % SEQ LA      Inodemap
```

```
  Inode    = LFS.Inode           % SEQ RECORD Inode
  LB       = LFS.LB              % SEQ RECORD Log Block
  F        = LFS.F               % IN         Rep. of a File
  Log      = LFS.Log             % SEQ Byte   Log
  LEC      = LFS.LEC             % UNION      Log Entry Content
  PLog     = LFS.PLog            % SEQ RECORD Parsed Log

VAR
  dir     := Dir{}
  log     := Log{}
  MaxDirect                      % constant max. number of direct log blocks
                                 % in an inode (eg. 10 direct log blocks)
  inodeMap,
  incache : IN -> Inode          % inode cache
  cache   : LA -> Data           % log block content cache
```

`DiskCache` implements the disk block cache; it looks for a log block in the cache before reading it from the log.

```
FUNC DiskCache (la) -> Data =
        cache!la => RET cache(la)
    [*] RET (LFS.ReadLog(log, la) AS LB).content
```

`InodeCache` implements the inode cache; it looks for the inode in the cache before reading it according to the inodemap.

```
FUNC InodeCache (in) -> Inode =
        incache!in => RET incache(in)
    [*] RET LFS.ReadLog(log, inodemap[in]) AS Inode

% The invariant says that if we give the cached disk to UnixFile.Flatten,
% the resulting inodes are the same as the inodes defined by the log
% according to the abstraction function in LFS.
% INVARIANT ( ALL in | DiskCache!in ==>
%                        UnixFile.Flatten(DiskCache, InodeCache(in)) =
%                        LFS.UpdatedFlatten(log, in) )
```

`Crash` restores the cache and incache to reflect the current state in the log in order to maintain the invariant.

```
PROC Crash() =                          % modelled on LFS.UpdateFlatten
VAR plog0 := ParseLog(log),
    plog := plog0.sub(                  % the part of plog after
        LFS.FindInodeMap(plog), plog.size) | % the last inodemap
```

```
  inodemap := plog[0],                   % the last inodemap in plog
  % construct most updated seq of blocks by examining plog
  DO plog # {} => VAR la, lec |
    (la, lec) := plog.head;
    plog := plog.tail;
    lec IS LB =>                          % a newly-written block is found
    BEGIN
      VAR in := lec.inum |
        lec.bnum < MaxDirect =>           % For a direct log block make sure
          incache(in) := InodeCache(in);  % there's an inode cache entry,
          incache(in).beg[lec.bnum] := la % then update it.
        [*] BEGIN VAR la', j |            % For an indirect log block,
        % choose the index block address la' and index j in this block
        % that should be updated to the la we got from the log entry.
              cache(la') := UpdateInd(DiskCache(la), i, la);
              UnixFile.Flatten(DiskCache, InodeCache(in))[lec.bnum] = la =>
              SKIP
            END                           % can't get here unless we chose
                                          % la' and j correctly
    END
    [*] SKIP
  OD
```

Update a single entry in an indirect block with a new log address

```
FUNC UpdateInd(data, i, la) -> Data = ...

...

END CacheLFS
```