Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 31 Revised October 25, 1993

Sequential Transactions with Caching

There are many situations in which we want to make a "big" action atomic, either with respect to concurrent execution of other actions (everyone else sees that the big action has either not started or run to completion) or with respect to failures (after a crash the big action has either not started or run to completion).

Some examples:

Debit/credit: $x := x + \Delta$; $y := y - \Delta$
Reserve airline seat
Rename file
Allocate file and update freespace information
Schedule meeting: room and six participants
Prepare report on one million accounts

Why is atomicity important? There are two main reasons:

Stability: A large persistent state is hard to fix it up if it gets corrupted. Manual fixup is impractical, and ad-hoc automatic fixup is too hard to code correctly. Atomicity is a valuable automatic fixup mechanism.

Consistency: We want the state to change one big action at a time. This has several advantages:

It's easy for the client to recover if the server storing the state crashes.

The state remains consistent if the client crashes.

Concurrent clients always see a consistent state, that is, one that satisfies the invariant of the system. It's much easier to code the client correctly if you can count on this invariant.

Here we treat failures without concurrency; a later handout treats concurrency without failures. A grand unification is possible and is left as an exercise, since the two constructions are more or less orthogonal.

We begin by repeating the `SequentialTransaction` spec and the `LogRecovery` implementation from Handout 10, with some refinements. Then we give a much more efficient implementation which allows for caching data; this is usually necessary for decent performance. Unfortunately it complicates matters considerably. We give a rather abstract version of the caching implementation, and then sketch the concrete specialization that is in common use. Finally we discuss some pragmatic issues.

There are four levels of failure:

Transaction abort: not really a failure, in the sense that no work is lost except by request.

Crash: the volatile state is lost, but the only effect is to abort all uncommitted transactions.

Media failure: the stable base is lost, but it is recovered from the permanent log, so that the effect is the same as a crash.

Catastrophe or disaster: the stable base and the permanent log are both lost, leaving the system in an undefined state.

The specification

```
MODULE NaiveSequentialTransactions [
  V,                                          % Value of an action
  S WITH { s0: ()->S },                       % State
  A WITH { meaning: A->S->(V, S) }            % Action
  ]

VAR
  ss        := S$s0()                         % stable state
  vs        := S$s0()                         % volatile state

PROC Do(a) -> V = << VAR v | (v, vs) := a.meaning(vs); RET v >>
PROC Commit() = << ss := vs >>

PROC Crash () = << vs := ss >>                % "aborts" the transaction

END NaiveSequentialTransactions
```

Here is a simple example, with variables X and Y constituting the stable state, and x and y the volatile state.

Action	X	Y	x	y
	5	5	5	5
Do$(x := x - 1)$; *Do*$(y := y + 1)$				
	5	5	4	6
Commit				
	4	6	4	6
crash before commit				
	5	5	5	5

If we want to take account of the possibility that the server (specified by this module) may fail separately from the client, then the client needs a way to detect this. Otherwise a server failure and restart after the decrement of x in the example could result in $X = 5, Y = 6$, because the client will continue with the decrement of y and the `Commit`. Alternatively, if the client fails at that point, restarts, and repeats its actions, the result would be $X = 3, Y = 6$. To avoid these problems we add another piece of state, a `ph` variable that keeps track of the phase of the transaction, and a `Begin` action that marks the start of a big action as well at the `Commit` action that marks its end.

```
MODULE SequentialTransactions [
  V,                                          % Value of an action
  S WITH { s0: ()->S },                       % State
  A WITH { meaning: A->S->(V, S) }            % Action
  ]

VAR
  ss       := S$s0()                          % stable state
  vs       := S$s0()                          % volatile state
  ph       : ENUM[idle, run] := idle          % phase (volatile)
EXCEPTION crash

PROC Begin() = << vs := ss; ph := run >>

PROC Do(a) -> V RAISES {crash} =
   VAR v |
     << ph = run => (v, vs) := a.meaning(vs); RET v
                [*] RAISE crash >>

PROC Commit() RAISES {crash} =
   << ph = run => ss := vs; ph := idle
              [*] RAISE crash >>

PROC Crash () = << vs := ss, ph := idle >>  % "aborts" the transaction

END SequentialTransactions
```

Here is the previous example extended with the `ph` variable.

Action	X	Y	x	y	ph
	5	5	5	5	idle
Begin();					
Do$(x := x - 1)$;					
Do$(y := y + 1)$					
	5	5	4	6	run
Commit					
	4	6	4	6	idle
Crash before Commit					
	5	5	5	5	idle

Uncached implementation

Next we give the simple uncached implementation based on logging.

```
MODULE LogRecovery [                              % implements SequentialTr
   V,                                             % Value of an action
   S0 WITH { s0: ()->S0 },                        % State
   A WITH { meaning: A->S->(V, S) }               % Action
   ] =

TYPE
  U        =  S -> S                              % atomic Update
  L        =  SEQ U                               % Log
  S        =  S0 WITH { "+":=DoLog }              % State with useful ops

VAR
  ss       := S.s0()                              % stable state
  vs       := S.s0()                              % volatile state
  sl       := L{}                                 % stable log
  vl       := L{}                                 % volatile log
  ph       : ENUM[idle, run] := idle              % phase (volatile)
EXCEPTION crash

% ABSTRACTION to SequentialTransaction
   ss = ss + sl
   vs = vs
   ph = ph

% INVARIANT
   vs = ss + sl + vl                              % except during Crash
```

```
APROC Begin() = << vs := ss; sl := {}; vl := {}; ph := run >>

APROC Do(a) -> V RAISES {crash} = << VAR v, l |
   ph = run /\ (v, vs + l) = (a.meaning)(vs) =>
       vs := vs + l; vl := vl + l; RET v
   [*] RAISE crash >>

PROC Commit() RAISES {crash} =
   ph=run => << sl := vl; ph := idle >> ; Redo()
         [*] RAISE crash

PROC Crash() = VAR s |
   vs := ss; vl := {}; ph := idle;          % what the crash does
   vl := sl; Redo(); vs := ss               % the recovery action

PROC Redo() =                               % replay vl, then clear sl
   DO vl # {} => << ss := ss + vl.head >>; << vl := vl.tail >> OD
   << sl := {} >>

FUNC DoLog(s, l) -> S =                     % s+l = DoLog(s, l)
      l = {} => RET s                       % apply U's in l to s
   [*] RET DoLog((l.head)(s), l.tail))

END LogRecovery
```

Here is the previous example in this implementation, assuming that `A = U`. You may wish to apply the abstraction function to the state at each point and check that each action simulates an action of the spec.

Action	X	Y	x	y	`sl`	`vl`	`ph`
	5	5	5	5	$\{\}$	$\{\}$	`idle`
Begin(); *Do*$(x := x - 1)$; *Do*$(y := y + 1)$							
	5	5	4	6	$\{\}$	$\{x := 4; y := 6\}$	`run`
Commit							
	5	5	4	6	$\{x := 4; y := 6\}$	$\{x := 4; y := 6\}$	`idle`
Redo: apply $x := 4$							
	4	5	4	6	$\{x := 4; y := 6\}$	$\{y := 6\}$	`idle`
Redo: apply $y := 6$							
	4	6	4	6	$\{x := 4; y := 6\}$	$\{\}$	`idle`
Redo: erase `sl`							
	4	6	4	6	$\{\}$	$\{\}$	`idle`
Crash before commit							
	5	5	5	5	$\{\}$	$\{\}$	`idle`

Log idempotence

For this redo crash recovery to work, we need idempotence of logs: `s + l + l = s + l`, since a crash can happen during recovery. From this we get (remember that “<=” on sequences is “prefix”)

```
l1 <= l2 ==> s + l1 + l2 = s + l2,
```

and more generally

```
(ALL l1, l2 | Hiccups(l1, l2) ==> s + l1 + l2 = s + l2)
```

where

```
Hiccups(l1, l2)->BOOL =                    % l1 a seq of tries at l2
  RET l1 = {}
     \/ (EXISTS l, l'| l1 = l+l' /\ l#{} /\ l<=l2 /\ Hiccups(l', l2))
```

because

```
Hiccups(l1, l2) /\ l1 # {} ==> l1 = l11 + l12 /\ l12<=l2
```

Hence `s+l1+l2 = s+l11+l12+l2 = s+l11+l2` because `s'+l12+l2=s'+l2`.

We can get log idempotence if the `U`’s commute and are idempotent, or if they are all writes like the assignments to x and y in the example, or writes of disk blocks. Often, however, we want to make more general updates atomic, for instance, inserting an item into a page of a a B-tree. We can make general `U`’s log idempotent by attaching a UID to each one and recording it in `S`:

```
TYPE  S = [ss, tags: SET UID]
      U = [uu: SS->SS, tag: UID] WITH { meaning:=Meaning }

FUNC Meaning(u, s)->S =
   u.tag IN s.tags => RET s
   [*] RET S{ ss := (u.uu)(s.ss), tags := s.tags ++ u.tag }
```

If all the `u`'s in `l` have different tags, we get log idempotence. The way to think about this is that the modified updates have the meaning: if the tag isn't already in the state, do the original update, otherwise don't do it.

A practical implementation of this arranges that each `U` operates on a single variable (that is, maps one value to another without looking at any other part of `S`; in the usual application, a variable is one disk block). It assigns a version number `VN` to each `U` and keeps the `VN` of the most recently applied `U` with each block. Then you can tell whether a `U` has already been applied just by comparing its `VN` with the `VN` of the block. For example, if the version number of the update is `11`:

```
             Original    Idempotent
The disk     x:  Int     x:  Int
block                    vn:  Int
The update   x := x + 1  vn = 10 =>x := x + 1
                                   vn := 11
                         [*] SKIP
```

Note: `vn = 10` implies that exactly updates `1`, `2`, . . . , `10` have been applied.

Writing the log atomically

This implementation is still not practical, because it requires writing the entire log atomically, and it might be bigger than the one disk block that the disk hardware writes atomically. There are various ways to get around this, but the standard one is to add a stable sph variable which can be idle or commit. We view `LogRecovery` as a spec for our new implementation, in which the `sl` of the spec is empty unless `sph = commit`. The module below includes only the parts that are different from `LogRecovery`.

```
MODULE IncrementalLog                          % implements LogRecovery

...

VAR ...
  sph      : ENUM[idle, commit]         := idle    % stable phase
  vph      : ENUM[idle, run, commit]    := idle    % volatile phase

% ABSTRACTION to LogRecovery
```

```
    sl = (sph = commit => sl  [*] {})
    ph = (sph # commit => vph [*] idle)
    the identity elsewhere

APROC Begin() = << vs := ss; sl := {}; vl := {}; vph := run >>

APROC Do(a) -> V RAISES {crash} = << VAR v, l |
    vph = run /\ ...                                % rest as before

PROC Commit() RAISES {crash} =
    vph = run =>
      % copy vl to sl a bit at a time
      VAR l := vl |
         DO l # {} => << sl := sl ++ l.head; l := l.tail >> OD;
      << sph := commit; vph := commit >>;
      Redo()
    [*] RAISE crash

PROC Crash() = VAR s |
    vs := s; vl := {}; vph := idle;                % what the crash does
    vph := sph; vl := (vph = idle => {} [*] sl);  % the recovery action
    Redo(); vs := ss

PROC Redo() =                                 % replay vl, then clear sl
    DO vl # {} => << ss := ss + vl.head >>; << vl := vl.tail >> OD
    DO sl # {} => << sl := sl.tail >> OD;
    << sph := idle; vph := idle >>

END IncrementalLog
```

And here is the example again.

Action	X	Y	x	y	sl	vl	sph	vph
Begin;Do;Do								
	5	5	4	6	$\{\}$	$\{x := 4; y := 6\}$	idle	run
Commit								
	5	5	4	6	$\{x := 4; y := 6\}$	$\{x := 4; y := 6\}$	commit	commit
Redo:$x := 4; y := 6$								
	4	6	4	6	$\{x := 4; y := 6\}$	$\{\}$	commit	commit
Redo:cleanup								
	4	6	4	6	$\{\}$	$\{\}$	idle	idle

We have described `sph` as a separate stable variable, but in practice each transaction is labeled with a unique transaction identifier, and `sph = commit` for a given transaction is represented by the presence of a *commit record* in the log that includes the transaction's identifier. Conversely, `sph = idle` is represented by the absence of a commit record or by the presence of a later "transaction end" record in the log. The advantages of this representation are that writing `sph` can be batched with all the other log writes, and that no storage management is needed for the `sph` variables of different transactions.

Aborts

The `SequentialTransaction` spec *aborts* the transaction, discarding its updates and turning it into `SKIP`, whenever there is a crash before the commit. It's a small and useful extension to add an `Abort` procedure to the spec, so that the client can choose to abandon the transaction.

```
APROC Abort () = << vs := ss; ph := idle >>  % same effect as Crash
```

Caching

We would like to have an implementation of `SequentialTransaction` that can run fast. To this end it should:

> Allow the volatile state `vs` to be cached so that the frequently used parts of it are fast to access, but not require a complete copy of the parts that are the same in the stable state.
>
> Decouple `Commit` from actually applying the updates that have been written to the stable log, because this slows down `Commit`, and it also does a lot of random disk writes that do not make good use of the disk. By waiting to write out changes until the main memory space is needed, we have a chance of accumulating many changes to a single disk block and paying only one disk write to record them all.
>
> Decouple crash recovery from actually applying updates. This is important once we have decoupled `Commit` from applying updates, since a lot of updates can now pile up and recovery can take a long time. Also, we get it more or less for free.

So our implementation has a stable representation of a state, which we call a *base*, and the actual stable state is the base plus the updates in the stable log. Thus the base may not include all the updates that are in the stable state. `Commit` need only write the updates to the stable log, since this gets them into the stable state; a background process updates the stable base. We keep the volatile state up to date so that `Do` can return its result quickly. The price paid in performance for this scheme is that we have to reconstruct the volatile state from the stable base and the log after a crash, rather than reading it directly from the stable state which no longer exists. So there's an incentive to limit the amount by which the background process runs behind.

Normally the volatile state consists of entries in the cache. Although the abstract implementation below does not make this explicit, the cache usually contains versions of disk blocks as they will appear when all the updates have been done. Thus the stable base is updated simply by writing out disk blocks from the cache. The cache together with the stable base represents the volatile state. (The cache is usually called a "buffer pool" in the database literature, where these techniques originated.)

We want to flush parts of the cache to the base independently of what is committed; otherwise we might run out of cache space if there are transactions that don't commit for a long time. Thus the base may run *ahead* of the stable state as well as behind. This means that the stable log must include "undo" operations that can be used to reverse the uncommitted updates in case the transaction aborts instead of committing.

Although we don't want to be forced to keep the base up with the log, we do want to discard old log entries after they have been applied to the base, whether or not the transaction has committed, so the log space can be reused. Of course, log entries for undos can't be discarded until `Commit`.

Finally, we want to be able to keep discarded log entries forever in a "permanent log" so that we can recover the base in case it is corrupted by a media failure. The permanent log is usually kept on magnetic tape.

Here is a summary of our requirements:

Complete permanent log for media recovery.

Crash recovery log that can be truncated.

Separate undo log to simplify truncating the crash recovery log.

Buffer pool that can be flushed independently of locking or commits.

The `LogAndCache` implementation below is a full description of a practical transaction system, except that it doesn't deal with concurrency or with distributing the transaction at multiple sites. The strategy is to:

Factor the state into independent components, for example, disk blocks.

Factor the actions into updates, and cache the updates. Each update is not only atomic but works on only one state component. Updates for different components commute.

Define an *undo* action for each action (not update). The action followed by its undo leaves the state unchanged.

Keep separate log and undo log, both stable and volatile.

Log	:	sequence of updates
UndoLog	:	sequence of undo actions

Usually `W` (a cached update) is just `b(ba) := d` (that is, set the contents of a block of the base to a new value), as described in `BufferPool` below. This is classical caching, and it may be helpful to bear it in mind as a concrete implementation of these ideas, which is worked out in detail in `BufferPool`. Note that this kind of caching has another important property: we can get the current value of `b(ba)` from the cache. This property isn't essential for correctness, but it certainly makes it easier for `Do` to be fast.

```
MODULE LogAndCache [                          % implements SequentialTr
  V,                                          % Value of an action
  S0 WITH {s0:=()->S0},                       % abstract State
  A  WITH {meaning: A->(S->(V, S))},          % Action
  B0 WITH {toS: B0->S0, SToB: S0->B0}         % stable Base
  ] =

TYPE
  S        =  S0 WITH {"+":=DoLogOnS,         % State with ops
                       toB:=B0.StoB}
  B        =  B0 WITH {"+" := DoLog,          % Base with useful ops
                      "++":= DoCache,
                      "-" := UndoLog}
  U        =  B -> B                          % Update
  Un       =  A                               % Undo
  W        =  B -> B                          % update in cache
  L        =  SEQ U                           % Log
  UL       =  SEQ Un                          % Undo Log
  C        =  SET W                           % Cache

VAR    % constants
  cancel   :  Un                              % cancel the latest
                                              % uncancelled Un
  mark     :  U                               % distinct identity fcn
  commit   :  U                               % distinct identity fcn
  tags     := {mark, commit}

VAR
  sb       := S.s0().toB                      % Stable Base
  c        :  C := {}                         % Cache (dirty part)

  pl       := L{}                             % Permanent Log

  sl       := L{}                             % Stable Log
  sul      := UL{}                            % Stable Undo Log

  vl       := L{}                             % Volatile Log
  vul      := UL{}                            % Volatile Undo Log

  forcing  : Bool := false                    % In Force()?
  vph      : ENUM[idle,run] := idle
```

Note that there are two logs, called `L` and `UL` (for undo log). An `L` records updates; the difference between an update `U` and an action `A` is that an action can be an arbitrary state change, while an update can be applied atomically to the stable base. Thus a single action must in general be broken down into several updates. For example, if the base is just raw disk pages, the only atomic operation is to write a single disk page, so an update must be small enough that it changes only one page.

`UL` records undo actions `Un` which reverse the effect of actions. These are actions, not updates; a `Un` is converted into a suitable sequence of `U`'s when it is applied. When we apply an undo, we treat it like any other action, except that it doesn't need an undo itself; this is because we only apply undos to abort a transaction, and we never change our minds about aborting a transaction. We do need to ensure either that each undo is applied only once, or that undo actions have log idempotence. Since we don't require log idempotence for ordinary actions (only for updates), it's unpleasant to require it for undos. Instead, we arrange to remove each undo action from the undo log atomically with applying it. We implement this idea with a special `Un` called `cancel` which means: don't apply the next earlier `Un` in the log that hasn't already been cancelled, and we write a `cancel` to `vul/sul` atomically as we write the updates of the undo action to `vl/sl`. For example, after $\mathtt{un}_1$, $\mathtt{un}_2$, and $\mathtt{un}_3$ have been processed, `ul` would be

$$
\begin{aligned}
& \mathtt{un}_0\ \mathtt{un}_1\ \mathtt{un}_2\ \mathtt{cancel}\ \mathtt{un}_3\ \mathtt{cancel}\ \mathtt{cancel} \\
= \; & \mathtt{un}_0\ \mathtt{un}_1\ \mathtt{un}_2\ \mathtt{cancel}\ \mathtt{cancel} \\
= \; & \mathtt{un}_0\ \mathtt{un}_1\ \mathtt{cancel} \\
= \; & \mathtt{un}_0
\end{aligned}
$$

Of course many other encodings are possible, but this one is simple and fits in well with the rest of the implementation.

Examples of `A`'s:

```
f(x) := y                          % simple overwriting
f(x) := f(x) + y                   % not idempotent
f := f{x -> }                      % delete
split node
```

This implementation has commit records in the log rather than a separate `sph` variable as in `IncrementalLog`. This makes it easier to have multiple transactions in the stable log. In the logs we also introduce a `mark` record to delimit the updates of a single action.

We have the following requirements on `U` and `W`

Adding an `L` to `sl` is atomic (this isn't really a constraint, because it can easily be arranged either by using a commit record, or by the method described in 1990's Problem Set 2).

Applying a `W` to `sb` is atomic.

A `W` looks only at a small part of `sb` when it is applied.

Mapping `U` to `W` is cheap and requires looking only at a small part of `sb`, at least most of the time.

U's have the log idempotence property described in LogRecovery. Note that A's don't need to have this property, since they don't appear in a log, and Un's don't need to have it because of the special cancel trick.

W's commute: w1 + w2 = w2 + w1.

```
% ABSTRACTION to SequentialTransaction
   ss = sb + sl - sul
   vs = ss + vl

% INVARIANTS
   [1] (ALL l1, l2 | sl = l1 ++ commit + l2   % Stable undos cancel
                     /\ ~commit IN l2.asSet   % uncommitted tail of sl
                     ==> sb + l1 = sb + sl - sul )

   [2] ~forcing ==> (sb+sl = sb+sl+vl-vul) % Volatile undos cancel vl

   [3] ss + vl = sb ++ c                   % Cache is correct

   [4] S.s0().toB + pl + sl = ss           % Permanent log is complete

% External interface

PROC Begin() = << vph = run => Abort() [*] vph := run >>

PROC Do(a) -> V RAISES {crash} =
  << vph = run => RET DoOrUndoA(a, AToUn(a, sb++c))
             [*] RAISE crash >>

PROC Commit() RAISES {crash} =
  vph = run => ForceAll();
               << sl := sl ++ commit; sul := {}; vph = idle>>
           [*] RAISE crash

PROC Abort () = Undo(); ForceAll(); vph := idle

PROC Checkpoint() = VAR c' := c |            % move sl+vl to pl
   DO VAR w | w IN c' => Flush(w); c' := c' - {w} OD;
   Truncate()

PROC Crash() = << vl := {}; vul := {}; c := {} >>;
               Redo(); Undo(); vph := idle
```

```
% Internal procedures invoked from Do and Commit

PROC DoOrUndoA(a, un) -> V = <<            % also invoked from Undo
   VAR  v, l, vs := sb ++ c |
    (v, l) := AToL(a, vs);
    vl := vl + l + mark; vul := vul ++ un;
    VAR c' | vs++c' = vs+l => c := c + c';  % Find w's for action a
    RET v >>

PROC ForceAll() = DO vl # {} => Force() OD

PROC Force() =                             % move one a from vl to sl
   VAR l1, l2 |                            % l1 = first action
      vl = l1 + l2 /\ OneActionUpdates(l1) =>
         forcing := true;
         << sl := sl + l1; sul := sul + vul.head >>;
         vl := l2; vul := vul.tail;
         forcing := false

% Internal procedures invoked from Crash or Abort

PROC Redo() = VAR l := sl |                % Restore vs after crash
   DO l # {} => VAR vs := sb ++ c, w |     % Find w for each u in l
       vs ++ {w} = vs + L{l.head} => c := c ++ w; l := l.tail
   OD

PROC Undo() =                              % Apply sul+vul to vs
   VAR ul := sul + vul, i := 0 |
       DO ul # {} => VAR un := ul.last |
           ul := ul.reml;
           un=cancel => i := i+1 [*] i>0 => i := i-1 [*]
           DoOrUndoA(un, cancel)
       OD;
   % Every entry in sul+vul has a cancel,
   %  and everything is undone in vs.

% Background actions to flush updates from cache to base
%  and to truncate sl

THREAD Background() =
   DO
```

```
      VAR w | sb # sb ++ {w} => Flush(w)
   [] DropW()
   [] Truncate()
   OD

PROC Flush(w) =                               % Apply w to sb; does WAL
  DO sb ++ {w} + sl # sb + sl => Force() OD; % Terminates by I3
  << sb := sb ++ {w} >>

PROC DropW() = VAR w | w IN c /\ sb ++ {w} = sb => c := c - {w}

PROC Truncate() = VAR l1, l2 |                 % Move some of sl to pl
   sl = l1 + l2 /\ sb + l2 = sb + sl => << pl := pl + l1; sl := l2 >>

% Media recovery
```

The idea is to reconstruct the stable base `sb` from the permanent log `pl` by redoing all the updates, starting with a fixed initial `sb`. Details are left as an exercise.

```
% Miscellaneous functions

FUNC AToL(a, b) -> (V, L) = VAR v, l |
   (v, (b + l).toS) = a.meaning(b.toS) /\ l.asSet * tags = {} =>
   RET (v, l)

FUNC AToUn(a, b) -> Un = VAR un, v, s' |
   (v, s') = a.meaning(b.toS) /\ (nil, b.toS) = un.meaning(s')
   => RET un

FUNC OneActionUpdates(l) -> Bool =
   RET (l.last IN tags /\ l.reml.asSet * tags = {})

% These procedures are for assertions, not for the
   implementation.

FUNC DoLog(b, l) -> B =                       % b+l = DoLog(b, l)
      l = {} => RET b                         % apply U's in l to b
   [*] RET DoLog((l.head)(b), l.tail))

FUNC DoLogOnS(s, l) -> S = RET (s.toB + l).toS  % s+l = DoLogOnS(s, l)

FUNC DoCache(b, c) -> B =                     % b++c = DoCache(b, c)
   DO VAR w | w IN c => b := w(b), c := c -- w OD; RET b
```

```
FUNC UndoLog(b, ul) -> B =                        % b-l = UndoLog(b, l)
   ul = {} => RET b
   []  VAR ul1, un, ul2 | ul = ul1++un + ul2++cancel
                          /\ ul2.asSet = {cancel} /\ un # cancel =>
          RET UndoLog(b, ul1 + ul2)
   [] VAR ul1, un | ul = ul1 ++ un /\ un # cancel =>
          RET UndoLog(un(b.toS).toB, ul1)

END LogAndCache
```

We can summarize the ideas in `LogAndCache`:

> Writing stable state before committing a transaction requires undo. We need to do this because cache space for changed state is limited, while the size of a transaction may not be limited.
>
> Every uncommitted log entry has a logged undo. The entry and the undo are made stable by a single atomic action (using some low-level coding trick that we leave as an exercise for the reader).
>
> Recovery is complete redo followed by undo of uncommitted transactions. Because of the complete redo, undos are always from a clean state, and hence can be actions.
>
> An undo is executed like a regular action, i.e, logged. The undo of the undo is a special `cancel` action.

The standard implementation of the ideas in `LogAndCache` is to make a `U` and a `W` work on a single block of data. The `W` just gives the current value of the block, and the `U` maps one such block into another. Both blocks and `U`s carry sequence numbers so that we can get the log idempotence property without restricting the kind of mapping that a `U` does, using the method described earlier in this handout.

The simplest case is a block equal to a single disk block, for which we have an atomic write. Often a block is chosen to be several disk blocks to reduce the size of indexes and improve the efficiency of disk transfers. In this case care must be taken that the write is still atomic; many commercial systems get this wrong.

The following module is incomplete.

```
MODULE BufferPool [                          % implements LogAndCache
  V,                                         % Value of an action
  S0 WITH {s0:=()->S0},                      % abstract State
  A WITH { meaning: A->(S->[v, s]) },        % Action
  D                                          % Data
  ] =

TYPE
```

```
  SN        =  INT                              % Sequence Number
  BA        =  INT                              % Block Address
  LB        =  [sn, d]                          % Logical Block

  B         =  BA -> LB                         % Base
  U         =  [sn, ba, f: LB->LB]              % Update
  Un        =  A
  W         =  [ba, lb]                         % update in cache

  L         =  SEQ U
  UL        =  SEQ Un
  C         =  SET W

FUNC vb(ba) -> LB = VAR w IN c | w.ba = ba => RET w.lb [*] RET sb(ba)
   % This is the standard abstraction function for a cache
```

`w.ba # u.ba =>` `w` commutes with `u`, because `u` only looks at `u.ba`.

```
(ALL b | (ALL ba | ba # u.ba => u(b).ba = b(ba)
         (ALL b′ | b*u, b(u.ba) = b′(u.ba) => u(b)(u.ba)=u(b′)(u.ba)))
```

So the guard on Flush is just

```
    (EXISTS u | u IN vl /\ u.ba = w.a)
```

because in `Do` we get `w` as `W{ba:=u.ba, lb:=u(vb)(u.ba)}`.

```
END BufferPool
```

Transactions Meet the Real World

Various problems arise in using the transaction abstraction we have been discussing to implement actual transactions such as ATM withdrawals or airline reservations. We mention the most important ones briefly.

The most serious problems arise when a transaction includes changes to state that is not completely under the computer's control. An ATM machine, for example, dispenses cash; once this has been done, there's no straightforward way for a program to take back the cash, or even to be certain that the cash was actually dispensed. So neither undo not log idempotence may be possible. Changing the state of a disk block has neither of these problems.

So the first question is: *Did it get done*? The jargon for this is "testability". Carefully engineered systems do as much as possible to provide the computer with feedback about what happened in the real world, whether it's dispensing money, printing a check, or unlocking a door. This means having sensors that are independent of actuators and have a high probability of being able to detect whether or not an intended physical state transition occurred. It also means designing the computer-device interface so that after a crash the computer can test whether the device received and performed a particular action; hence the name “testability”.

The second question is: *Is undo possible*? The jargon for this is “compensation”. Carefully engineered systems include methods for undoing at least the important effects of changes in the state of the world. This might involve executing some kind of compensating transaction, for instance, to reduce the size of the next order if too much is sent in the current one, or to issue a stop payment order for a check that was printed erroneously. Or it might require manual intervention, for instance, calling up the bank customer and asking what really happened in yesterday’s ATM transaction. Usually compensation is not perfect.

Because compensation is complicated and imperfect, the first line of defense against the problems of undo is to minimize the probability that a transaction involving real-world state changes will have to abort. To do this, break it into two transactions. The first runs entirely inside the computer, and it makes all the internal state changes as well as posting instructions for the external state changes that are required. The second is as simple as possible; it just executes the posted external changes. Often the second transaction is thought of as a message system that is responsible for reliably delivering an action message to a physical device, and also for using the testability features to ensure that the action is taken exactly once.

The other major difficulty in transactions that interact with the world arises only with concurrent transactions. It has to do with input: if the transaction requires a round trip from the computer to a user and back it might take a long time, because users are slow and easily distracted. For example, a reservation transaction might accept a travel request, display flight availability, and ask the user to choose a flight. If the transaction is supposed to be atomic, seats on the displayed flights must remain available until the user makes her choice, and hence can’t be sold to other customers. To avoid these problems, systems usually insist that a single transaction begin with user input, end with user output, and involve no other interactions with the user. So the reservation example would be broken into two transactions, one inquiring about flight availability and the other attempting to reserve a seat. A later handout on concurrent transactions discusses this issue in more detail.