Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 — Principles of Computer Systems

Handout 18 March 31, 1997

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 either has not started or has run to completion).

Some examples:

Debit/credit: $x := x + \Delta$; $y := y - \Delta$
Reserve airline seat
Rename file
Allocate file and update free space 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:

- When the server storing the state crashes, it's easy for the client to recover.
- When the client crashes, the state remains consistent.
- 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.

The simplest way to use the atomicity of transactions is to start each transaction with no volatile state. Then there is no need for an invariant that relates the volatile state to the stable state between atomic actions. Since these invariants are the trickiest part of easy concurrency, getting rid of them is a major simplification.

Overview

In this handout we treat failures without concurrency; handout 19 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 can classify failures into four levels. We show how to recover from the first three.

> 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.

We begin by repeating the `SequentialTransaction` spec and the `LogRecovery` code from handout 7, with some refinements. Then we give much more efficient code that 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 code, and then sketch the concrete specialization that is in common use. Finally we discuss some pragmatic issues.

The specification

An `A` is an encoded action, that is, a transition from one state to another that also returns a result value. The `meaning` method decodes an `A` and yields the transition function. Note that an `A` is a function, that is, a deterministic transition.

```
MODULE NaiveSequentialTr [
    V,                                              % Value of an action
    S WITH { s0: ()->S },                           % State
    A WITH { meaning: A->S->(V, S) }                % Action
    ] EXPORT Do, Commit, Crash =

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

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

APROC Commit() = << ss := vs >>

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

END NaiveSequentialTr
```

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);				
	5	5	4	5
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 introduce a new `Begin` action to mark the start of a transaction as `Commit` marks the end. We use another state variable `ph` (for phase) that keeps track of whether there is an uncommitted transaction in progress. A transaction in progress is aborted whenever there is a crash, or if another `Begin` action is invoked before it commits. We also introduce an `Abort` action so the client can choose to abandon a transaction explicitly.

This interface is slightly redundant, since `Abort = Begin; Commit`, but it's the standard way to do things. Note that `Crash = Abort` also; this is not redundant, since the client can't call `Crash`.

```
MODULE SequentialTr [
     V,                                                   % Value of an action
     S WITH { s0: ()->S },                                % State
     A WITH { meaning: A->S->(V, S) }                     % Action
     ] EXPORT Begin, Do, Commit, Abort, Crash =

VAR
     ss          := S.s0()                                % Stable State
     vs          := S.s0()                                % Volatile State
     ph          : ENUM[idle, run] := idle                % PHase (volatile)

EXCEPTION crashed

APROC Begin() = << vs := ss; ph := run >>                 % aborts any current trans.

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

APROC Commit() RAISES {crashed} =
     << IF ph = run => ss := vs; ph := idle [*] RAISE crashed FI >>

PROC Abort () = << vs := ss, ph := idle >>                % same as Crash
PROC Crash () = << vs := ss, ph := idle >>                % 'aborts' the transaction

END SequentialTr
```

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

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

Uncached code

Next we give the simple uncached code based on logging. Note that `ss` is not the same as the `ss` of `SequentialTr`; the abstraction function gives the relation between them.

This code may seem impractical, since it makes no provision for caching the volatile state `vs`. We will study how to do this caching in general later in the handout. Here we point out that a scheme very similar to this one is used in Lightweight Recoverable Virtual Memory[1], with copy-on-write used to keep track of the differences between `vs` and `ss`.

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

TYPE U          =  S -> S                             % atomic Update
     L          =  SEQ U                              % Log
     S          =  S0 WITH { "+":=DoLog }             % State with useful ops
     Ph         =  ENUM[idle, run]                    % PHase

VAR  ss         := S.s0()                             % stable state
     vs         := S.s0()                             % volatile state
     sl         := L{}                                % stable log
     vl         := L{}                                % volatile log
     ph         := idle                               % phase (volatile)
     rec        := false                              % recovering

EXCEPTION crashed

% ABSTRACTION to SequentialTr
     SequentialTr.ss = ss + sl
     SequentialTr.vs = (~ rec => vs [*] rec => ss + sl)
     SequentialTr.ph = ph
```

[1] M. Satyanarayanan et al., Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems* **12**, 1 (Feb. 1994), pp 33-57.

```
% INVARIANT
     ~ rec ==> vs = ss + sl + vl
     (EXISTS l | l <= vl /\ ss + sl = ss + l)
     % Applying sl to ss is equivalent to applying a prefix l of vl. That is, the
     % state after a crash will be the volatile state minus some updates at the end.

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

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

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

PROC Abort() = << vs := ss; sl := {}; vl := {}; ph := idle >>

PROC Crash() =
     << vs := ss; vl :={}; ph := idle; rec := true >>; % what the crash does
     vl := sl; Redo(); vs := ss; rec := false         % the recovery action

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

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

END LogRecovery
```

Here is what this code does for the previous example, assuming for simplicity 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 | IsHiccups(l1, l2) ==> s + l1 + l2 = s + l2)
```

where

```
FUNC IsHiccups(k: Log, l) -> Bool =
% k is a sequence of attempts to complete l
     RET    k = {}
         \/ (EXISTS k', l'|    k = k' + l' /\ l' # {} /\ l' <= l
                             /\ IsHiccups(k', l) )
```

because we can keep absorbing the last hiccup `l'` into the final complete `l`. See handout 7 for more detail.

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

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.

Practical code for 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 block	`x: Int`	`x:  Int` `vn: Int`

The update	`x := x + 1`	`IF  vn = 10 => x := x + 1;`
		`               vn := 11`
		`[*] SKIP FI`

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

Writing the log atomically

This code is still not practical, because it requires writing the entire log atomically in `Commit`, 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 that can be `idle` or `commit`. We view `LogRecovery` as a spec for our new code, in which the `sl` of the spec is empty unless `sph = commit`. The module below includes only the parts that are different from `LogRecovery`. It changes `sl` only one update at a time.

```
MODULE IncrementalLog                               % implements LogRecovery

...

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

% ABSTRACTION to LogRecovery
    LogRecovery.sl = (sph = commit => sl  [*] {})
    LogRecovery.ph = (sph # commit => vph [*] idle)
    the identity elsewhere

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

APROC Do(a) -> V RAISES {crashed} = <<
    IF  vph = run …                                  % the rest as before

PROC Commit() =
    IF 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 crashed
    FI

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

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.

Note that you still have to be careful about the order of disk writes: all the log data must really be on the disk before `sph` is set to `commit`. This complication is below the level of abstraction of this discussion.

Caching

We would like to have code for `SequentialTr` 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.

Our code 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 thread 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 code below does not make this explicit, the cache usually contains the more recent of variables, that is, the values they have when all the updates have been done. Thus the stable base is updated simply by writing out variables from the cache. If the write operations write complete disk blocks, as is most often the case, it's convenient for the cached variables to be disk blocks also. If the variables are smaller, you have to read a disk block before writing it; this is called an 'installation read'. The advantage of smaller variables, of course, is that they take up less space in 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 'install' parts of the cache to the base independently of what is committed (for a processor cache, install is usually called 'flush', and for a file system cache it is sometimes called 'sync'). Otherwise we might run out of cache space if there are transactions that don't commit for a long time. Even if all transactions are short, a popular part of the cache might always be touched by a transaction that hasn't yet committed, so we couldn't install it and therefore couldn't truncate the log. 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. In order to keep undoing simple when the abort is caused by a crash, we arrange things so that before applying an undo, we use the stable log to completely do the action that is being undone. Hence an undo is always applied to an "action consistent" state, and we don't have to worry about the interaction between a undo and the smaller atomic updates that together comprise the action. To implement this rule we need to add an action's updates and its undo to the log atomically.

To be sure that we can abort a transaction after installing some parts of the cache to the stable base, we have to follow the "write ahead log" or WAL rule, which says that before a cache entry can be installed, all the actions that affected that entry and all their undo's must be in the stable log.

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:

Cache that can be installed independently of locking or commits.

Crash recovery log that can be truncated.

Separate undo log to simplify truncating the crash recovery log.

Complete permanent log for media recovery.

The `LogAndCache` code 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 concrete code for 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.

Although in our examples an update is usually a write of one disk block, the code works on top of any kind of atomic update, for example a B-tree or even another transactional system. The latter case implements each `U` as a transaction in terms of updates at a still lower level. Of course this idea can be applied recursively to yield a *n* level system. This is called 'multi-level recovery'.[2] It's possible to make a multi-level system more efficient by merging the logs from several levels into a single sequential structure.

Why would you want to complicate things in this way instead of just using a single-level transaction, which would have the same atomicity? There are at least two reasons:

- The lower level may already exist in a form that you can't change, without the necessary externally accessible locking or commit operations. A simple example is a file system, which typically provides atomic file renaming, on which you can build something more general. Or you might have several existing database systems, on top of which you want to build transactions that change more than one database. We show in handout 27 how to do this using two-phase commit. But if the existing systems don't implement two-phase commit, you can still build a multi-level system on them.

- Often you can get more concurrency by allowing lower level transactions to commit and release their locks. For example, a B-tree typically holds locks on whole disk blocks to maintain the integrity of the index data structures, but at the higher level only individual entries need to be locked.

---

[2] D. Lomet. MLR: A recovery method for multi-level systems. *Proc. SIGMOD Conf.*, May, 1992, pp 185-194.

```
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
    ] = EXPORT Begin, Do, Commit, Abort, Crash =

TYPE
    S           =  S0 WITH {"+":=DoLogOnS,              % State with ops
                            toB:=B0.StoB}
    B           =  B0 WITH {"+" := DoLog,               % Base with useful ops
                           "++":= DoCache,
                           "-" := UndoLog}
    Tag         =  ENUM[commit]
    U           =  B -> B                               % Update
    Un          =  (A + ENUM[cancel])                   % Undo
    W           =  B -> B                               % update in cache
    L           =  SEQ (SEQ U + Tag)                    % Log
    UL          =  SEQ Un                               % Undo Log
    C           =  SET W WITH {"++":=CombineCache}      % Cache
    Ph          =  ENUM[idle, run]                      % Phase

VAR
    sb          := S.s0().toB                           % Stable Base

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

    c           :  C := {}                              % Cache (dirty part)
    vl          := L{}                                  % Volatile Log
    vul         := UL{}                                 % Volatile Undo Log

    vph         := idle                                 % Volatile PHase

    pl          := L{}                                  % Permanent Log

    undoing     := false
```

Note that there are two logs, called `L` and `UL` (for undo log). A `L` records groups of 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. All the updates from one action form one group in the log. The reason for grouping the updates of an action is that, as we have seen, we have to add all the updates of an action, together with its undo, to the stable log atomically. There are various ways to represent a group, for example as a sequence of updates delimited by a special `mark` token, but we omit these details.

`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` that means: don't apply the next earlier `Un` in the log that hasn't already been canceled, and we write a `cancel` to `vul/sul` atomically as we write the updates of the undo action to `vl/sl`. For example, after `un1`, `un2`, and `un3` have been processed, `ul` would be

```
  un0 un1 un2 cancel un3 cancel cancel
= un0 un1 un2 cancel cancel
= un0 un1 cancel
= un0
```

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

Examples of `A`'s:

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

This code 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. For brevity we omit the machinations with `vph`.

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

- We can add atomically to `sl` both all the `U`'s of an action and the action's undo (`ForceOne` does this).
- Applying a `W` to `sb` is atomic (`Install` does this).
- A `W` looks only at a small part of `b` when it is applied, normally only one component (`DoCache` does this).
- Mapping `U` to `W` is cheap and requires looking only at a small part (normally only one component) of `sb`, at least most of the time (`Apply` does this).
- `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.
- All `W`'s in the cache commute. This ensures that we can install cache entries in any order (`CombineCache` assures this).

```
% ABSTRACTION to SequentialTr
    SequentialTr.ss = sb.toS + sl – sul
    SequentialTr.vs = (~undoing => sb.toS + sl + vl [*] sb.toS + (sl+vl)–(sul+vul)
    SequentialTr.ph = (~undoing => vph idle)


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

```
    [2] sb + sl = sb + sl + vl - vul                % Volatile undos cancel vl

    [3] ~ undoing ==> vs.toB = sb ++ c              % Cache is correct

    [4] (ALL w1 :IN cache, w2 :IN cache |           % All W's in cache commute
            w1 + w2 = w2 + w1).

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

% External interface

PROC Begin() = << IF vph = run => Abort() [*] SKIP FI; vph := run >>

PROC Do(a) -> V RAISES {crashed} = <<
    IF   vph = run => VAR v | v := Apply(a, AToUn(a, sb ++ c)); RET v
    [*]  RAISE crashed
    FI >>

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

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

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

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

% Internal procedures invoked from Do and Commit

APROC Apply(a, un) -> V = <<                        % called by Do and Undo
    VAR v, l, vb := sb ++ c |
        (v, l) := AToL(a, vb);                      % find U's that do a
        vl := vl + l; vul := vul + {un};
        VAR c' | sb ++ c ++ c' = sb ++ c + l        % Find w's for action a
            => c := c ++ c';
        RET v >>

PROC ForceAll() = DO vl # {} => ForceOne() OD; RET

APROC ForceOne() = << VAR l1, l2 |                  % move one a from vl to sl
    sl := sl + {vl.head}; sul := sul + {vul.head};
    vl := vl.tail; vul := vul.tail >>

% Internal procedures invoked from Crash or Abort

PROC Redo() = VAR l := + : sl |                     % Restore c from sl after cras
    DO << l # {} => VAR vb := sb ++ c, w |          % Find w for each u in l
        vb ++ {w} = vb + 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;
            IF un=cancel => i := i+1 [*] i>0 => i := i-1 [*] Apply(un, cancel) FI
        OD;
    % Every entry in sul + vul has a cancel, and everything is undone in vs.
```

```
    undoing := false
```

% Background actions to install updates from `cache` **to base and to truncate** `sl`

```
THREAD Background() =
    DO
        VAR w | sb # sb ++ {w} => Install(w)
    []  Drop()
    []  Truncate()
    []  SKIP
    OD

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

APROC Drop() = << VAR w :IN c | sb ++ {w} = sb => c := c - {w} >>

APROC 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 |                 % all U's in one group
    l.size = 1 /\ (v, (b + l).toS) = a.meaning(b.toS) => 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
```

The remaining functions are only used in guards and invariants.

```
FUNC DoLog(b, l) -> B =                                % b + l = DoLog(b, l)
    IF  l = {} => RET b                                % apply U's in l to b
    [*] VAR us := l.head |
        RET DoLog((   us IS Tag \/ us = {} => b
                  [*] (us.head)(b), {us.tail} + l.tail))
    FI

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 :IN c | b := w(b), c := c - {w} OD; RET b

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

A cache is a set of commuting update functions. When we combine two caches `c1` and `c2`, we want the total effect of `c1` and `c2`, and all the updates still have to commute and be atomic updates. The `DoCache` below just states this requirement, without saying how to achieve it. Usually it's done by requiring updates that don't commute to compose into a single update that is still atomic. In the usual case updates are writes of single variables, which do have this property, since `u1 * u2 = u2` if both are writes of the same variable.

```
FUNC CombineCache(c1, c2) -> C =                          % c1++c2 = CombineCache(c1,c2)
     VAR c |    (* : c.seq) = (* : c1.seq) * (* : c2.seq)
             /\ (ALL w1 :IN c, w2 :IN c | w1 # w2 ==> w1 * w2 = w2 * w1) => RET c

END LogAndCache
```

We can summarize the ideas in `LogAndCache`:

> Writing stable state before committing a transaction requires undo. We need to write before committing because cache space for changed state is limited, while the size of a transaction may not be limited, and also to avoid livelock that keeps us from installing some cache entries.
>
> 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). We must log an action and its undo before installing a cache entry affected by it; this is write ahead logging.
>
> 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, that is, logged. The undo of the undo is a special `cancel` action.

Buffer pools

The standard code for 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 `W`'s (that is, cache 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; these are called 'log sequence numbers' or LSN's in the database literature.

The LSN's are also used to implement the WAL guard in `Install` and the guard in `Truncate`. It's OK to install a `W` if the LSN of the last entry in `sl` is at least as big as the n of the `W`. It's OK to drop a `U` from the front of `sl` if every uninstalled `W` in the cache has a bigger LSN.

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
    ] EXPORT ... =

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
```

The essential property is that updates to different blocks commute: `w.ba # u.ba ==> w` commutes with `u`, because `u` only looks at `u.ba`. Stated precisely:

```
(ALL b |    (ALL ba | ba # u.ba ==> u(b)(ba) = b(ba)
        /\ (ALL b’ | b(u.ba) = b’(u.ba) ==> u(b)(u.ba) = u(b’)(u.ba)) )
```

So the guard on `Install` 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 nor 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. Handout 19 on concurrent transactions discusses this issue in more detail.