19. 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:

1. *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.
2. *Consistency*: We want the state to change one big action at a time, so that between changes it is always 'consistent', that is, it always satisfies the system's invariant and always reflects exactly the effects of all the big actions that have been applied so far. 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 state 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 20 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 state 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 state and the permanent log are both lost, leaving the system in an undefined state.

We begin by repeating the `SequentialTr` spec and the `LogRecovery` code from handout 7 on file systems, 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 spec

An `A` is an encoded action, that is, a transition from one state to another that also returns a result value. Note that an `A` is a function, that is, a deterministic transition.

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

TYPE A          =  S->(V, S)                            % Action

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

APROC Do(a) -> V = << VAR v | (v, vs) := a(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 as 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; s0 initially
    ] EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A          =  S->(V, S)                            % Action

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

EXCEPTION crashed

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

APROC Do(a) -> V RAISES {crashed} = <<
    IF ph = run => VAR v | (v, vs) := a(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; it is basically the same as the `LogRecovery` module of handout 7 on file systems, with the addition of `ph`. 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
    ] EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A          =  S->(V, S)                            % Action
     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
```

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

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

% 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(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 makes each `U` operate on a single variable (that is, map 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 the log 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, called "write-ahead log" or 'WAL' in the database literature, 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:

1. 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.
2. 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.
3. 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.
4. Allow uncommitted updates to be written to the stable log, and even applied to the stable state. This saves a lot of bookkeeping to keep track of which parts of the cache go with uncommitted transactions, and it allows a transaction to make more updates that will fit in the cache.

Our new caching code has a stable state; as in `LogRecovery`, the committed state is the stable state plus the updates in the stable log. Unlike `LogRecovery`, the stable state may not include all the committed updates. `Commit` need only write the updates to the stable log, since this gets them into the abstract stable state `SequentialTR.ss`; a `Background` thread updates the concrete stable state `LogAndCache.ss`. 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 state and the log after a crash, rather than reading it directly from the committed 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 most recent values of variables, that is, the values they have when all the updates have been done. Thus the stable state 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 state 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 stable state independently of what is committed (for a processor cache, install is usually called 'flush', and for a file system cache it is usually 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 stable state may run *ahead* of the committed 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 an 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 state, 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 therefore all their undo's) must be in the stable log.

Although we don't want to be forced to keep the stable state up with the log, we do want to discard old log entries after they have been applied to the stable state, whether or not the transaction has committed, so the log space can be reused. Of course, log entries for undo's 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 stable state 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 (or 'redo') 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 (see handout 20) or with distributing the transaction among multiple `SequentialTr` modules (see handout 27). The strategy is to:

- Factor the state into independent components, for example, disk blocks.
- Factor the actions into log updates called `U`'s and cache updates called `W`'s. Each cache update not only is atomic but works on only one state component. Cache updates for different components commute. Log updates do not need either of these properties.
- 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 (not updates)

The essential step is installing a cache update into the stable state. This is an internal action, so it must not change the abstract stable or volatile state. As we shall see, there are many ways to satisfy this requirement.

Usually `W` (a cached update) is just `s(ba) := d` (that is, set the contents of a block of the stable state 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 `s(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; S0 initially
     ] = EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A          =  S->(V, S)                                % Action
     S          =  S0 WITH {"+":=DoLog,                     % State with ops
                            "++":= DoCache,
                            "-" := UndoLog}
     Tag        =  ENUM[commit]
     U          =  S -> S                                   % Update
     Un         =  (A + ENUM[cancel])                       % Undo
     W          =  S -> S                                   % 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  ss         := S.s0()                                   % Stable State

     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 must interact properly with a cache update `W`. To achieve this, a single action must in general be broken down into several updates. 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.

A cache update `W` must be applied atomically to the stable state. For example, if the stable state 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` that 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 undo's 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 undo's. Instead, we arrange to remove each undo action from the undo log atomically with applying it. We code 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` might 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 `sph`.

We have the following requirements on `U`, `Un`, 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 `ss` is atomic (`Install` does this).
- Applying a `W` to `ss` doesn't change the abstract `ss` or `vs`. This is a key property that replaces the log idempotence of `LogRecovery`.
- A `W` looks only at a small part of `s` 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 `ss`, at least most of the time (`Apply` does this).
- 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 = ss + sl – sul
    SequentialTr.vs = (~undoing => ss + sl + vl [*] ss + (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
                      ==> ss + l1 = ss + sl - sul )

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

    [3] ~ undoing ==> ss + sl + vl = ss ++ c           % Cache is correct; this is vs

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

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

    [6] (ALL w :IN c |  ss ++ {w} + sl                 % Any cache entry can be installed
                      = ss        + sl
```

% 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, ss ++ 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(); vph := idle

PROC Checkpoint() = VAR c' := c, w |                  % move sl + vl to pl
    DO c' # {} => w := Install(); c' := c' - {w} OD;  % until everything in c' is installed
    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, vs := ss ++ c |
        (v, l) := AToL(a, vs);                        % find U's that do a
        vl := vl + l; vul := vul + {un};
        VAR c' | ss ++ c ++ c' = ss ++ c + l          % Find w's for action a
            => c := c ++ c';
        RET v >>
```

```
PROC ForceAll() = DO vl # {} => ForceOne() OD; RET     % more all of vl to sl

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 crash
    DO << l # {} => VAR vs := ss ++ 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;
            IF un=cancel => i := i+1 [*] i>0 => i := i-1 [*] Apply(un, cancel) FI
        OD; undoing := false
    % Every entry in sul + vul has a cancel, and everything is undone in vs.
```

% Background actions to install updates from c to ss and to truncate sl

```
THREAD Background() =
    DO
        Install()
    []  ForceOne()                                     % Enough of these implement WAL
    []  Drop()
    []  Truncate()
    []  SKIP
    OD

APROC Install() -> W = << VAR w :IN c |                % Apply some w to ss; requires WAL
       ss # ss ++ {w}                                  % w isn't already in ss
    /\ ss ++ {w} + sl = ss + sl =>                     % w is in sl, the WAL condition
        ss := ss ++ {w}; RET w >>

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

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

% Media recovery

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

% Miscellaneous functions

```
FUNC AToL(a, s) -> (V, L) = VAR v, l |                 % all U's in one group
    l.size = 1 /\ (v, s + l) = a(s) => RET (v, l)

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

The remaining functions are only used in guards and invariants.

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

FUNC DoCache(s, c) -> S =                             % s ++ c = DoCache(s, c)
    DO VAR w :IN c | s := w(s), c := c - {w} OD; RET s

FUNC UndoLog(s, ul) -> S =                            % s - l = UndoLog(s, l)
    IF   ul = {} => RET s
    []   ul.last # cancel => RET UndoLog((u.last)(s), ul.reml)
    []   VAR ul1, un, ul2 | un # cancel /\ ul = ul1 + {un, cancel} + ul2 =>
         RET UndoLog(s, 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, undo's 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.
- Writing a `W` to stable state doesn't change the abstract stable state. This means that redo recovery works. It's a strong constraint on the relation between logged `U`'s and cached `W`'s.

Multi-level recovery

Although in our examples an update is usually a write of one disk block, the `LogAndCache` code works on top of any kind of atomic update, for example a B-tree or even another transactional system. The latter case codes each `W` 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.

Buffer pools

The standard code for the ideas in `LogAndCache` makes a `U` and a `W` read and write 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 code 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.

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

```
MODULE BufferPool [                                % implements LogAndCache
    V,                                             % Value of an action
    S0 WITH {s0:=()->S0}                           % abstract State
    Data
    ] EXPORT ... =

TYPE A          =  S->(V, S)                       % Action
    SN          =  Int                             % Sequence Number
    BA          =  Int                             % Block Address
    LB          =  [sn, data]                      % Logical Block

    S           =  BA -> LB                        % State
    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 vs(ba) -> LB = VAR w IN c | w.ba = ba => RET w.lb [*] RET ss(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 s |   (ALL ba | ba # u.ba ==> u(s)(ba) = s(ba)
       /\ (ALL s' | s(u.ba) = s'(u.ba) ==> u(s)(u.ba) = u(s')(u.ba)) )
```

So the guard in `Install` testing whether `w` is already installed is just
`    (EXISTS u | u IN vl /\ u.ba = w.a)`
because in `Do` we get `w` as `W{ba:=u.ba, lb:=u(vs)(u.ba)}`.

```
END BufferPool
```

Transactions meet the real world

Various problems arise in using the transaction abstraction we have been discussing to code 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 20 on concurrent transactions discusses this issue in more detail.

20. Concurrent Transactions

We often (usually?) want more from a transaction mechanism than atomicity in the presence of failures: we also want atomicity in the presence of concurrency. As we saw in handout 14 on practical concurrency, the reasons for wanting to run transactions concurrently are slow input/output devices (usually disks) and the presence of multiple processors on which different transactions can run at the same time. The latter is especially important because it is a way of taking advantage of multiple processors that doesn't require any special programming. In a distributed system it is also important because separate nodes need autonomy.

Informally, if there are two transactions in progress concurrently (that is, the `Begin` of one happens after the `Begin` and before the `Commit` of the other), we want all the observable effects to be as though all the actions of one transaction happen either before or after all the actions of the other. This is called *serializing* the two transactions; it is the same as making each transaction into an atomic action. This is good for the usual reason: it allows the clients to reason about each transaction separately as a sequential program. The clients only have to worry about concurrency in between transactions, and they can use the usual method for doing this: find invariants that each transaction establishes when it commits and can therefore assume when it begins.

Here is the standard example. We are maintaining bank balances, with `deposit`, `withdraw`, and `balance` transactions. The first two involve reading the current balance, adding or subtracting something, making a test, and perhaps writing the new balance back. If the read and write are atomic actions, then the sequence `read1`, `read2`, `write1`, `write2` will result in losing the effect of transaction 1. The third reads lots of balances and expects their total to be a constant. If its reads are interleaved with the writes of the other transactions, it may get the wrong total.

The other property we want is that if one transaction precedes another (that is, its `Commit` happens before the `Begin` of the other) then it is serialized first. This is sometimes called *external consistency*; it's not just a picky detail that only a theoretician would worry about, because it's needed to ensure that when you put two transaction systems together you still get a serializable system.

A piece of jargon you will sometimes see is that transactions have the ACID properties: Atomic, Consistent, Isolated, and Durable. Here are the definitions given in Gray and Reuter:

> **Atomicity**. A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.
>
> **Consistency**. A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program.
>
> **Isolation**. Even though transactions execute concurrently, it appears to each transaction T that others executed either before T or after T, but not both.

> **Durability**. Once a transaction completes successfully (commits), its changes to the state survive failures.

The first three appear to be different ways of saying that all transactions are serializable.

Many systems implement something weaker than serializability for committed transactions in order to allow more concurrency. The standard terminology for weaker degrees of isolation is degree 0 through degree 3, which is serializability. Gray and Reuter discuss the specs, code, advantages, and drawbacks of weaker isolation in detail (section 7.6, pages 397-419).

We give a spec for concurrent transactions. Coding this spec is called 'concurrency control', and we briefly discuss a number of coding techniques.

Spec

The spec is written in terms of the *histories* of the transactions: a history is a sequence of (action, result) pairs, called *events* below. The order of the events for a single transaction is fixed: it is the order in which the transaction did the actions. A spec must say that for all the committed transactions there is a total ordering with three properties:

> *Serializable*: Doing the actions in the total order would yield the same result from each action, and the same final state, as the results and final state actually obtained.
>
> *Externally consistent*: The total order is consistent with the partial order established by the `Begin`'s and `Commit`'s.
>
> *Non-blocking*: it's always possible to abort a transaction. This is necessary because when there's a crash all the active transactions must abort.

This is all that most transaction specs say. It allows anything to happen for uncommitted transactions. Operationally, this means that an uncommitted transaction will have to abort if it has seen a result that isn't consistent with any ordering of it and the committed transactions. It also means that the programmer has to expect completely arbitrary results to come back from the actions. In theory this is OK, since a transaction that gets bad results will not commit, and hence nothing that it does can affect the rest of the world. But in practice this is not very satisfactory, since programs that get crazy results may loop, crash, or otherwise behave badly in ways that are beyond the scope of the transaction system to control. So our spec imposes some constraints on how actions can behave even before they commit.

The spec works by keeping track of:

- The ordering requirements imposed by external consistency, in a relation `xc`.
- The histories of the transactions, in a map `y`.

It imposes an invariant on `xc` and `y` that is defined by the function `Invariant`. This function says that the committed transactions have to be serializable in a way consistent with `xc`, and that something must be true for the active transactions. As written, `Invariant` offers a choice of several "somethings"; the intuitive meaning of each one is described in a comment after its definition. The `Do` and `Commit` routines block if they can't find a way to satisfy the invariant. The invariant maintained by the system is `Invariant(committed, active, xc, y)`.

It's unfortunate that this spec deals explicitly with the histories of the transactions. Normally our specs don't do this, but instead give a state machine that only generates allowable histories. If there's a way to do this for the most general serializability spec I don't know what it is.

The function `Invariant` defining the main invariant appears after the other routines of the spec.

```
MODULE ConcurrentTransactions [
    V,                                          % Value
    S,                                          % State of database
    T                                           % Transaction ID
    ] EXPORT Begin, Do, Commit, Abort, Crash =

TYPE Result     =  [v, s]
    A           =  S -> Result                  % Action
    E           =  [a, v]                       % Event
    H           =  SEQ E                        % History

    TS          =  SET T                        % Transaction Set
    XC          =  (T, T)->Bool                 % eXternal Consistency
    TO          =  SEQ T                        % Total Order on T's
    Y           =  T -> H                       % state of transactions

VAR s0          :  S                            % current base state
    y           := Y{}                          % current transaction state
    xc          := XC{* -> false}               % current required XC

    active      :  TS{}                         % active transactions
    committed   :  TS{}                         % committed transactions
    installed   :  TS{}                         % installed transactions
    aborted     :  TS{}                         % aborted transactions
```

The sets `installed` and `aborted` are only for the benefit of careless clients; they ensure that `T`'s will not be reused and that `Commit` and `Abort` can be repeated without raising an exception.

Operations on histories and orderings

To define `Serializable` we need some machinery. A history `h` records a sequence of events, that is, actions and the values they return. `Apply` applies a history to a state to compute a new state; note that it fails if the actions in the history don't give back the results in the history. `Valid` checks whether applying the histories of the transactions in a given total order can succeed, that is, yield the values that the histories record. `Consistent` checks that a total order is consistent with a partial order, using the `closure` method (see section 9 of the Spec reference manual) to get the transitive closure of the external consistency relation and the <<= method for non-contiguous sub-sequence. Then `Serializable(ts, xc, y)` is `true` if there is some total order `to` on the transactions in the set `ts` that is consistent with `xc` and that makes the histories in `y` valid.

```
FUNC Apply(h, s) -> S =
    % return the end of the sequence of states starting at s and generated by
    % h's actions, provided the actions yield h's values. Otherwise undefined.
    RET {e :IN h, s' := s BY (e.a(s').v = e.v => e.a(s').s)}.last

FUNC Valid(y0, to) -> BOOL = RET Apply!( + : (to * y0), s0)
% the histories in y0 in the order defined by to are valid starting at s0

FUNC Consistent(to, xc0) -> BOOL =
    RET xc0.closure.set <= (\ t1, t2 | TO{t1, t2} <<= to).set

FUNC Serializable(ts, xc0, y0) -> BOOL =              % is there a good TO of ts
    RET ( EXISTS to | to.set = ts /\ Consistent(to, xc0) /\ Valid(y0, to) )
```

Interface procedures

A transaction is identified by a *transaction identifier* `t`, which is assigned by `Begin` and passed as an argument to all the other interface procedures. `Do` finds a result for the action `a` that satisfies the invariant; if this isn't possible the `Do` can't occur, that is, the transaction issuing it must abort or block. For instance, if concurrency control is coded with locks, the issuing transaction will wait for a lock. Similarly, `Commit` checks that the transaction is serializable with all the already committed transactions. `Abort` never blocks, although it may have to abort several transactions in order to preserve the invariant; this is called "cascading aborts" and is usually considered to be bad, for obvious reasons.

Note that `Do` and `Commit` block rather than failing if they can't maintain the invariant. They may be able to proceed later, after other transactions have committed. But some code can get stuck (for example, the optimistic schemes described later), and for these there must be a demon thread that aborts a stuck transaction.

```
APROC Begin() -> T =
% Choose a t and make it later in xc than every committed trans; can't block
    << VAR t | ~ t IN active \/ committed \/ installed \/ aborted =>
        y(t) := {}; active := active \/ {t}; xc(t, t) := true;
        DO VAR t' :IN committed | ~ xc.closure(t', t) => xc(t', t) := true OD;
        RET t >>

APROC Do(t, a) -> V RAISES {badT} =
% Add (a,v) to history; may block unless NC
    << IF  ~ t IN active => RAISE badT
       [*] VAR v, y' := y{t -> y(t) + {E{a, v}}} |
             Invariant(committed, active, xc, y') => y := y'; RET v >>

APROC Commit(t) RAISES {badT} = <<                     % may block unless AC (to keep invariant)
    IF  t IN committed \/ installed => SKIP            % repeating Commit is OK
    []  ~ t IN active \/ committed \/ installed => RAISE badT >>
    []  t IN active /\ Invariant(committed \/ {t}, active - {t}, xc, y) =>
        committed := committed \/ {t}; active := active - {t} >>

APROC Abort(t) RAISES {badT} = <<                      % doesn't block (need this for crashes)
    IF  t IN aborted => SKIP                           % repeating Abort is OK
    []  t IN active  =>
        % Abort t, and as few others as possible to maintain the invariant.
        % s is the possible sets of T's to abort; choose one of the smallest ones.
        VAR s      := {ts |   {t} <= ts /\ ts <= active
                           /\ Invariant(committed, active - ts, xc, y)},
            n      := {ts | ts IN s | ts.size}.min,
            aborts := {ts | ts IN s /\ ts.size = n}.choose |
                 aborted := aborted \/ aborts; active:= active - aborts;
                 y := y{t->}
    [*]  RAISE badT
    FI >>
```

Installation daemon

This is not really part of the spec, but it is included to show how the data structures can be cleaned up.

```
THREAD Install() = DO                                  % install a committed transaction in s0
    << VAR t |
             t IN committed
             % only if there's no other transaction that should be earlier
          /\ ( ALL t' :IN committed \/ active | xc(t , t') ) =>
                  s0 := Apply(y(t), s0);
                  committed := committed - {t}; installed := installed \/ {t}
                  % remove t from y and xc; this isn't necessary, but it's tidy
                  y := y{t -> };
                  DO VAR t' | xc(t , t') => xc := xc{(t , t') -> } OD;
    >>
    [*] SKIP
    OD
```

Function defining the main invariant

```
FUNC Invariant(com: TS, act: TS, xc0, y0) -> BOOL = VAR current := com + act |
         Serializable(com, xc0, y0)
    /\   % constraints on active transactions: choose ONE

AC       (ALL t :IN act |                     Serializable(com + {t}, xc0, y0) ))

CC                                            Serializable(com + act, xc0, y0)

EO       (ALL t :IN act | (EXISTS ts |
             com       <=ts /\ ts<=current /\  Serializable(ts  + {t}, xc0, y0) ))

OD       (ALL t :IN act | (EXISTS ts |
             AtBegin(t)<=ts /\ ts<=current /\  Serializable(ts  + {t}, xc0, y0) ))

OC1      (ALL t :IN act, h :IN Prefixes(y0(t)) | (EXISTS to, h1, h2 |
              to.set = AtBegin(t) /\ Consistent(to, xc0) /\ Valid(y0, to)
           /\ IsInterleaving(h1, {t' | t' IN current - AtBegin(t) - {t} | y0(t')})
           /\ h2 <<= h1      % subsequence
           /\ h.last.a(Apply(+ : (to * y0) + h2 + h.reml, s0) = h.last.v ))

OC2      (ALL t :IN act, h :IN Prefixes(y0(t)) | (EXISTS to, h1, h2, h3 |
              to.set = AtBegin(t) /\ Consistent(to, xc0) /\ Valid(y0, to)
           /\ IsInterleaving(h1, {t' | t' IN current - AtBegin(t) - {t} | y0(t')})
           /\ h2 <<= h1      % subsequence
           /\ IsInterleaving(h3, {h2, h.reml})
           /\ h.last.a(Apply(+ : (to * y0) + h3, s0) = h.last.v ))

NC       true

FUNC Prefixes(h) -> SET H = RET {h' | h' <= h /\ h' # {}}

FUNC AtBegin(t) -> TS = RET {t' | xc.closure(t', t)}
% The transactions that are committed when t begins.

FUNC IsInterleaving(h, s: SET H) -> BOOL =
% h is an interleaving of the histories in s. This is true if there's a
% multiset il that partitions h.dom, and each element of il extracts
% one of the histories in s from h
    RET (EXISTS il: SEQ SEQ Int |
         (+ : il) == h.dom.seq /\ {z :IN il | | z * h} == s.seq )
```

A set of transactions is serializable if there is a serialization for all of them. All versions of the invariant require the committed transactions to be serializable; hence a transaction can only commit if it is serializable with all the already committed transactions. There are different ideas about the uncommitted ones. Some ideas use `AtBegin(t)`: the transactions that committed before `t` started.

AC All Committable: every uncommitted transaction can commit now and `AC` still holds for the rest (implies that any subset of the uncommitted transactions can commit, since abort is always possible). Strict two-phase locking, which doesn't release any locks until commit, ensures `AC`.

CC Complete Commit: it's possible for all the transactions to commit (i.e., there's at least one that can commit and `CC` still holds for the rest). `AC ==> CC`. Two-phase locking, which doesn't acquire any locks after releasing one, ensures `CC`.

EO Equal Opportunity: every uncommitted transaction has some friends such that it can commit if they do. `CC ==> EO`.

OD Orphan Detection: every uncommitted transaction is serializable with its `AtBegin` plus some other transactions (a variation not given here restricts it to the `AtBegin` plus some other committed transactions). It may not be able to commit because it may not be serializable with all the committed transactions; a transaction with this property is called an 'orphan'. Orphans can arise after a failure in a distributed system when a procedure keeps running even though its caller has failed, restarted, and released its locks. The orphan procedure may do things based on the old values of the now unlocked data. `EO ==> OD`.

OC Optimistic Concurrency: uncommitted transactions can see some subset of what has happened. There's no guarantee that any of them can commit; this means that the code must check at commit. Here are two versions; `OC1` is stronger.

`OC1`: Each sees `AtBegin` + some other stuff + its stuff; this roughly corresponds to having a private workspace for each uncommitted transaction. `OD ==> OC1`.

`OC2`: Each sees `AtBegin` + some other stuff including its stuff; this roughly corresponds to a shared workspace for uncommitted transactions. `OC1 ==> OC2`

NC No Constraints: uncommitted transactions can see arbitrary values. Again, there's no guarantee that any of them can commit. `OC2 ==> NC`.

Note that each of these implies all the lower ones.

Code

In the remainder of the handout, we discuss various ways to code these specs. These are all ways to code the guards in `Do` and `Commit`, stopping a transaction either from doing an action which will keep it from committing, or from committing if it isn't serializable with other committed transactions.

Two-phase locking

The most common way to code this spec[1] is to ensure that a transaction can always commit (AC) by

> acquiring locks on data in such a way that the outstanding actions of active transactions always commute, and then
>
> doing each action of transaction `t` in a state consisting of the state of all the committed transactions plus the actions of `t`.

This ensures that we can always serialize `t` as the next committed transaction, since we can commute all its actions over those of any other active transaction. We proved a theorem to this effect in handout 17, the "big atomic actions" theorem. With this scheme there is at least one time where a transaction holds all its locks, and any such time can be taken as the time when the transaction executes atomically. If all the locks are held until commit (strict two-phase locking), *the serialization order is the commit order* (more precisely, the commit order is a legal serialization order).

To achieve this we need to associate a set of locks with each action in such a way that if two actions don't commute, then they have conflicting locks. For example, if the actions are just reads and writes, we can have a read lock and a write lock for each datum, with the rule that read locks don't conflict with each other, but a write lock conflicts with either. This works because two reads commute, while a read and a write do not. Note that the locks are on the actions, not on the updates into which the actions are decomposed to code logging and recovery.

Once acquired, `t`'s locks must be held until `t` commits. Otherwise another transaction could see data modified by `t`; then if `t` aborts rather than committing, the other transaction would also have to abort. Thus we would not be maintaining the invariant that every transaction can always commit, because the premature release of locks means that all the actions of active transactions may not commute. Holding the locks until commit is called *strict two-phase locking*.

A variation is to release locks before commit, but not to acquire any locks after you have released one. This is called *two-phase locking*, because there is a phase of acquiring locks, followed by a phase of releasing locks. Two-phase locking implements the CC spec.

One drawback of locking is that there can be deadlocks, as we saw in handout 14. It's possible to detect deadlocks by looking for cycles in the graph of threads and locks with arcs for the relations "thread `a` waiting for lock `b`" and "lock `c` held by thread `d`". This is usually not done for

[1] In Jim Gray's words, "People who do it for money use locks." This is not strictly true, but it's close.

mutexes, but it often is done by the lock manager of a database or transaction processing system, at least for threads and locks on a single machine. It requires global information about the graph, so it is expensive to code across a distributed system. The alternative is timeout: assume that if a thread waits too long for a lock it is deadlocked. Timeout is the poor man's deadlock detection; most systems use it. A transaction system needs to have an automatic way to handle deadlock because the clients are not supposed to worry about concurrency, and that means they are not supposed to worry about avoiding deadlock.

To get a lot of concurrency, it is necessary to have fine-granularity locks that protect only small amounts of data, say records or tuples. This introduces two problems:

There might be a great many of these locks.

Usually records are grouped into sets, and an operation like "return all the records with `hairColor = blue`" needs a lock that conflicts with inserting or deleting *any* such record.

Both problems are usually solved by organizing locks into a tree or DAG and enforcing the rule that a lock on a node conflicts with locks on every descendant of that node. When there are too many locks, *escalate* to fewer locks with coarser granularity. This can get complicated; see Gray and Reuter[2] for details.

We now make the locking scheme more precise, omitting the complications of escalation. Each lock needs some sort of name; we use strings, which might have the form `"Read(addr)"`, where `addr` is the name of a variable. Each transaction `t` has a set of locks `locks(t)`, and each action `a` needs a set of locks `protect(a)`. The `conflict` relation says when two locks conflict. It must have the property stated in invariant `I1`, that non-commuting actions have conflicting locks. Note that `conflict` need not be transitive.

Invariant `I2` says that a transaction has to hold a lock that protects each of its actions, and `I3` says that two active transactions don't hold conflicting locks. Putting these together, it's clear that all the committed transactions in commit order, followed by any interleaving of the active transactions, produces the same histories.

```
TYPE Lk          =  String
     Lks         =  SET Lk

CONST
     protect     :  A -> Lks
     conflict    :  (Lk, Lk) -> Bool

% I1: (ALL a1, a2 | a1 * a2 # a2 * a1 ==> conflict(protect(a1), protect(a2)))

VAR  locks       :  T -> Lks

% I2: (ALL t :IN active, e :IN y(t) | protect(e.a) <= locks(t))

% I3: (ALL t1 :IN active, t2 :IN active | t1 # t2 ==>
       (ALL lk1 :IN locks(t1), lk2 :IN locks(t2) | ~ conflict(lk1, lk2)))
```

[2]Gray and Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993, pp 406-421.

To maintain `I2` the code needs a partial inverse of the `locks` function that answers the question: does anyone hold a lock that conflicts with `lk`.

Multi-version time stamps

It's possible to give very direct code for the idea that the transactions take place serially, each one at a different instant —we make each one happen at a single instant of logical time. Define a logical time and keep with each datum a history of its value at every instant in time. This can be represented as a sequence of pairs (*time*, *value*), with the meaning that the datum has the given value from the given time until the time of the next pair. Now we can code `AC` by picking a time for each transaction (usually the time of its `Begin`, but any time between `Begin` and `Commit` will satisfy the external consistency requirement), and making every read obtain the value of the datum at the transaction's time and every write set the value at that time.

More precisely, a read at t gets the value at the next earlier definition, call it, and leaves a note that the value of that datum can't change between t and t' unless the transaction aborts. To maintain `AC` the read must block if t' isn't committed. If the read doesn't block, then the transaction is said to read 'dirty data', and it can't commit unless the one at t' does. This version implements `CC` instead of `AC`. A write at t may not be possible, because some other transaction has already read a different value at t. This is the equivalent of deadlock, because the transaction cannot proceed. Or, in Jim Gray's words, reads are writes (because they add to the history) and waits are aborts (because waiting for a write lock turns into aborting since the value at that time is already fixed).[3] These translations are not improvements, and they explain why multi-version time stamps have not become popular.

A drastically simplified form of multi-version time stamps handles the common case of a very large transaction `t` that reads lots of shared data but only writes private data. This case arises in running a batch transaction that needs a snapshot of an online database. The simplification is to keep just one extra version of each datum; it works because `t` does no writes. You turn on this feature when `t` starts, and the system starts to do copy-on-write for all the data. Once `t` is done (actually, there could be several), the copies can be discarded.

Optimistic concurrency control

It's easier to ask forgiveness than to beg permission.
Grace Hopper

Sometimes you can get better performance by allowing a transaction to proceed even though it might not be able to commit. The standard version of this *optimistic* strategy allows a transaction to read any data it likes, keeps track of all the data values it has read, and saves all its writes in local variables. When the transaction commits, the system atomically

checks that every datum read still has the value that was read, and

if this check succeeds, installs all the writes.

[3] Gray and Reuter, p 437.

This obviously serializes the transaction at commit time, since the transaction behaves as if it did all its work at commit time. If any datum that was read has changed, the transaction aborts, and usually retries. This implements OC1 or OC2. The check can be made efficient by keeping a version number for each datum. Grouping the data and keeping a version number for each group is cheaper but may result in more aborts.

The disadvantages of optimistic concurrency control are that uncommitted transactions can see inconsistent states, and that livelock is possible because two conflicting transactions can repeatedly restart and abort each other. With locks at least one transaction will always make progress as long as you choose the youngest one to abort when a deadlock occurs.

OCC can avoid livelock by keeping a private write buffer for each transaction, so that a transaction only sees the writes of committed transactions plus its own writes. This ensures that at least one uncommitted transaction can commit whenever there's an uncommitted transaction that started after the last committed transaction `t`. A transaction that started before `t` might see both old and new values of variables written by `t`, and therefore be unable to commit. Of course a private write buffer for each transaction is more expensive than a shared write buffer for all of them. This is especially true because the shared buffer can use copy-on-write to capture the old state, so that reads are not slowed down at all.

The Hydra design for a single-chip multi-processor[4] uses an interesting version of OCC to allow speculative parallel execution of a sequential program. The idea is to run several sequential segments of a program in parallel as transactions (usually loop iterations or a procedure call and its continuation). The desired commit order is fixed by the original sequential ordering, and the earliest segment is guaranteed to commit. Each transaction has a private write buffer but can see writes done by earlier transactions; if it sees any values that are later overwritten then it has to abort and retry. Most of this logic is coded by the hardware of the on-chip caches and write buffers.

Field calls and escrow locks

There is a specialization of optimistic concurrency control called "field calls with escrow locking" that can perform much better under some very special circumstances that occur frequently in practice. Suppose you have an operation that does

```
    << IF pred(v) => v := f(v) [*] RAISE error >>
```

where `f` is total. A typical example is a debit operation, in which `v` is a balance, `pred(v)` is `v > 100`, and `f(v)` is `v - 100`. Then you can attach to `v` a 'pending list' of the `f`'s done by active transactions. To do this update, a transaction must acquire an 'escrow lock' on `v`; this lock conflicts if applying any subset of the `f`'s in the pending list makes the predicate false. In general this would be too complicated to test, but it is not hard if `f`'s are increment and decrement (`v + n` and `v - n`) and `pred`'s are single inequalities: just keep the largest and smallest values that `v` could attain if any subset of the active transactions commits. When a transaction commits, you

[4] Hammond, Nayfeh, and Olukotun, A single-chip multiprocessor, *IEEE Computer*, Sept. 1997. Hammond, Willey, and Olukotun, Data speculation support for a chip multiprocessor, *Proc 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, Oct. 1998. See also http://www-hydra.stanford.edu/publications.shtml.

apply all its pending updates. Since these field call updates don't actually obtain the value of `v`, but only test `pred`, they don't need read locks. An escrow lock conflicts with any ordinary read or write. For more details, see Gray and Reuter, pp 430-435.

This may seem like a lot of trouble, but if `v` is a variable that is touched by lots of transactions (such as a bank branch balance) it can increase concurrency dramatically, since in general none of the escrow locks will conflict.

Full escrow locking is a form of locking, not of optimism. A 'field call' (without escrow locking) is the same except that instead of treating the predicate as a lock, it checks atomically at commit time that all the predicates in the transaction are still true. This *is* optimistic. The original form of optimism is a special case in which every `pred` has the form `v = old value` and every `f(v)` is just `new value`.

Nested transactions

It's possible to generalize the results given here to the case of *nested* transactions. The idea is that within a single transaction we can recursively embed the entire transaction machinery. This isn't interesting for handling crashes, since a crash will cause the top-level transaction to abort. It is interesting, however, for making it easy to program with concurrency inside a transaction by relying on the atomicity (that is, serializability) of sub-transactions, and for making it easy to handle errors by aborting unsuccessful sub-transactions.

With this scheme, each transaction can have sub-transactions within itself. The definition of correctness is that all the sub-transactions satisfy the concurrency control invariant. In particular, all committed sub-transactions are serializable. When sub-transactions have their own nested transactions, we get a tree. When a sub-transaction commits, all its actions are added to the history of its parent.

To code nested transactions using locking we need to know the conflict rules for the entire tree. They are simple: if two different transactions hold locks `lk1` and `lk2` and one is not the ancestor of the other, then `lk1` and `lk2` must not conflict. This ensures that all the actions of all the outstanding transactions commute except for ancestors and descendants. When a sub-transaction commits, its parent inherits all its locks.

Interaction with recovery

We do not discuss in detail how to put this code for concurrency control together with the code for recovery that we studied earlier. The basic idea, however, is simple enough: the two are almost completely orthogonal. All the concurrent transactions contribute their actions to the logs. Committing a transaction removes its undo's from the undo logs, thus ensuring that its actions survive a crash; the single-transaction version of recovery in handout 18 removes everything from the undo logs. Aborting a transaction applies its undo's to the state; the single-transaction version applies all the undo's.

Concurrency control simply stops certain actions (`Do` or `Commit`) from happening, and perhaps aborts some transactions that can't commit. This is clearest in the case of locking, which just prevents any undesired changes to the state. Multi-version time stamps use a more complicated representation of the state; the ordinary state is an abstraction given a particular ordering of the transactions. Optimistic concurrency control aborts some transactions when they try to commit. The trickiest thing to show is that the undo's that recovery does in `Abort` do the right thing.

Performance summary

Each of the coding schemes has some costs when everything is going well, and performs badly for some combinations of active transactions.

Locking pays the costs of acquiring the locks and of deadlock detection in the good case. Deadlocks lead to aborts, which waste the work done in the aborted transactions, although it's possible to choose the aborted transactions so that progress is guaranteed. If the locks are too coarse either in granularity or in mode, many transactions will be waiting for locks, which increases latency and reduces concurrency.

Optimistic concurrency control pays the cost of noticing competing changes in the good case, whether this is done by version numbers or by saving initial values of variables and checking them at `Commit`. If transactions conflict at `Commit`, they get aborted, which wastes the work they did, and it's possible to have livelock, that is, no progress, in the shared-write-buffer version; it's OK in the private-write-buffer version, since someone has to commit before anyone else can fail to do so.

Multi-version time stamps pay a high price for maintaining the multi-version state in the good case; in general reads as well as writes change it. Transaction conflicts lead to aborts much as in the optimistic scheme. This method is inferior to both of the others in general; it is practical, however, for the special case of copy-on-write snapshots for read-only transactions, especially large ones.

.