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. The simplest way for a program to ensure that it doesn't depend on anything except the invariants is to discard all state at the end of a transaction, and re-read whatever you need after starting the next transaction.

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 the largest 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, so that their execution does not overlap) 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 of all the events in the committed transactions that has three properties:

> *Serializable*: Doing the actions in the total order (starting from the initial state) 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. I don't know any way to do this for the most general serializability spec.

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: Action and result Value
    H           =  SEQ E                            % History

    TS          =  SET T                            % Transaction Set
    XC          =  (T, T)->Bool                     % eXternal Consistency; the first
                                                    % transaction precedes the second
    TO          =  SEQ T                            % Total Order on T's
    Y           =  T -> H                           % histories of transactions

VAR s0          :  S                                % current base state
    y           := Y{}                              % current transaction histories
    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 \/ := {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 \/ := {t}; 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 :IN s | | ts.size}.min,
            aborts := {ts :IN s | ts.size = n}.choose |
                aborted \/ := aborts; 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 - := {t}; 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 amultiset 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). Two-phase locking, which doesn't acquire any locks after releasing one, ensures `CC`. `AC ==> 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 any two actions that don't commute 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 *t′*, 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* is impossible if 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 work is done 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.

.