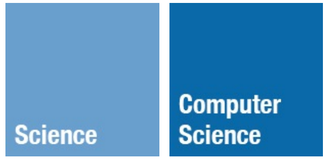
Alternative Concurrency Models



CS 450 : Operating Systems
Michael Lee <lee@iit.edu>

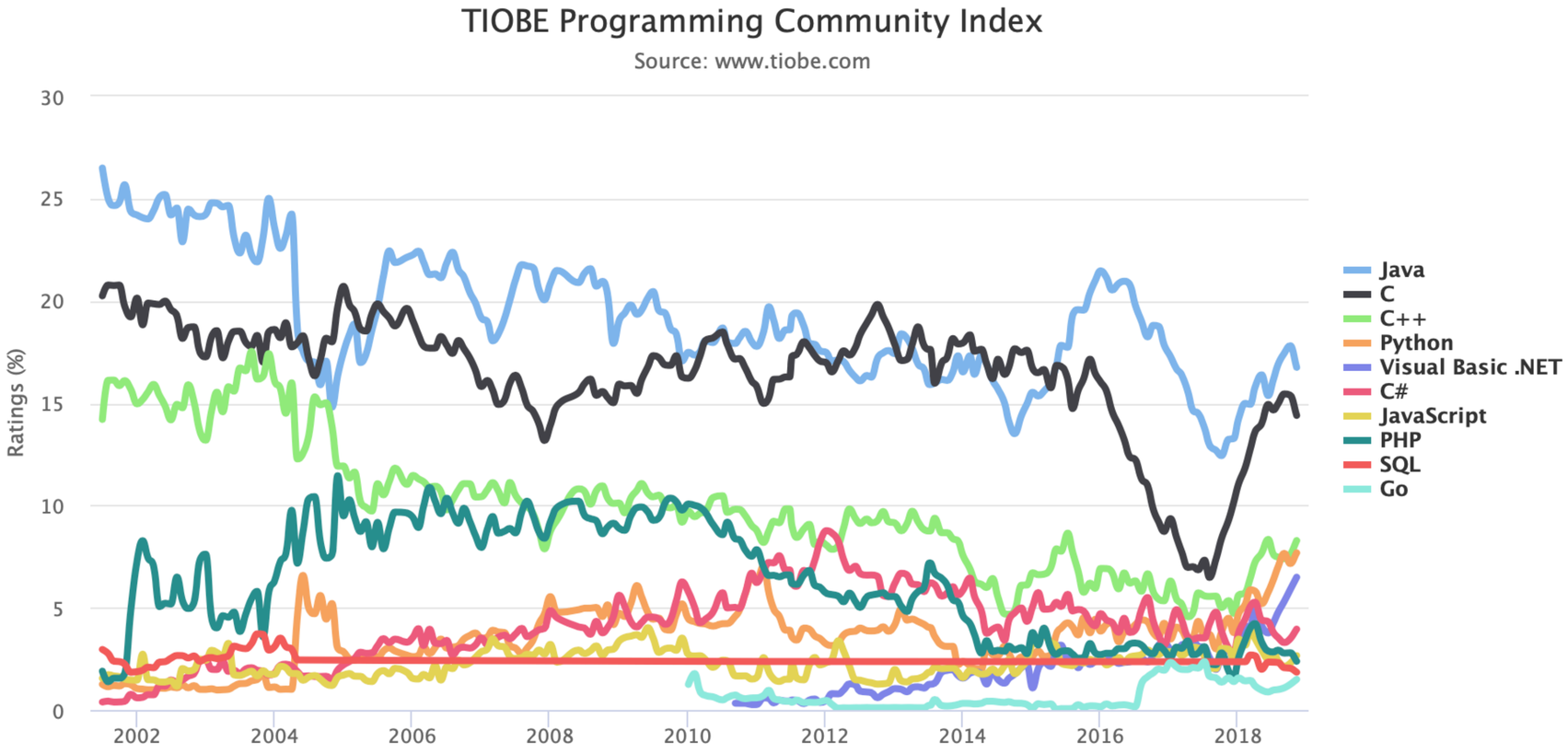
ILLINOIS INSTITUTE OF TECHNOLOGY
College of Science

"The free lunch is over. We have grown used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If we want our programs to run faster, we must learn to write parallel programs."

- Simon Peyton Jones, *Beautiful Concurrency*

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming …, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

- John Backus, Can Programming Be Liberated from the von Neumann Style? (1978)



TIOBE language popularity chart

most popular paradigms are

imperative and **object-oriented**

imperative: a program consists of a sequence of *statements* that read and alter process state

e.g.,
```
for (i=0; i<N; i++) {
    sum += arr[i];
}
```

early on, *procedural* languages helped us modularize imperative programs by separating logic into different procedures

… not quite good enough.

Bad programmers can too easily write “spaghetti code” (e.g., with globs & gotos)

OOP: bundle data and methods that act on them into *objects*; goal is *encapsulation*

e.g.,
```
acc1 = BankAccount(balance=1000.0)
acc2 = BankAccount(balance=0.0)
acc2.deposit(500.0)
acc1.transfer_to(acc2, 250.0)
print(acc1.balance(), acc2.balance())
```

In most OO languages, objects are *mutable*; i.e., objects may consist of many pieces of *shareable*, *changeable state*

(aka “big mutable balls”)

Most common concurrency model:

- explicitly created & managed *threads*
- *shared*, *freely mutable* state (memory)
- *lock*-based synchronization (e.g., semaphores, mutexes)

"Mutual-exclusion locks are one of the most widely used and fundamental abstractions for synchronization … Unfortunately, without specialist programming care, these benefits rarely hold for systems containing more than a handful of locks:

- *For correctness, programmers must ensure that threads hold the necessary locks to avoid conflicting operations being executed concurrently...*
- *For liveness, programmers must be careful to avoid introducing deadlock and, consequently, they may cause software to hold locks for longer than would otherwise be necessary ...*
- *For high performance, programmers must balance the granularity at which locking operates against the time that the application will spend acquiring and releasing locks."*

- Keir Fraser, *Concurrent Programming Without Locks*

implementing correct concurrent behavior via locks is hard!

but correctness can be verified via testing, right?

"… one of the fundamental problems with testing … [is that] testing for one set of inputs *tells you* nothing at all *about the behaviour with a different* set of inputs*. In fact the problem caused by state is typically worse — particularly when testing large chunks of a system — simply because even though the number of possible* inputs *may be very large, the number of possible* states *the system can be in is often even larger."*

"One of the issues (that affects both testing and reasoning) is the exponential rate at which the number of possible states grows — for every single bit *of state that we add we* double *the total number of possible states."*

- Ben Moseley and Peter Marks, *Out of the Tar Pit*

"Concurrency also affects testing … Running a test in the presence of concurrency with a known initial state and set of inputs tells you nothing at all *about what will happen the next time you run that very same test with the very same inputs and the very same starting state. . . and things can't really get any worse than that."*

- Ben Moseley and Peter Marks, *Out of the Tar Pit*

Another issue: *composability*

I.e., after building and testing a software module, can we easily combine it with other (tested) modules to build a system?

"... consider a hash table with thread-safe insert and delete operations. Now suppose that we want to delete one item A *from table* t1, *and insert it into table* t2*; but the intermediate state (in which neither table contains the item) must not be visible to other threads. Unless the implementor of the hash table anticipates this need, there is simply no way to satisfy this requirement… In short, operations that are individually correct (insert, delete) cannot be* ***composed*** *into larger correct operations."*

- Tim Harris et al, *Composable Memory Transactions*

lack of composability is a big problem!

- code modules can not make use of each other without additional reasoning/testing

"Civilization advances by extending the number of important operations which we can perform without thinking."

- Alfred North Whitehouse

the root problem is *shared, freely mutable state*

requires the use of *synchronization*

leading to unnecessary, or *accidental*, complexity in the implementation

"Anyone who has ever telephoned a support desk for a software system and been told to "try it again", or "reload the document", or "restart the pro- gram", or "reboot your computer" or "re-install the program" or even "re- install the operating system and then the program" has direct experience of the problems that state causes for writing reliable, understandable software."

- Ben Moseley and Peter Marks, *Out of the Tar Pit*

"Complexity is the *root cause of the vast majority of problems with soft- ware today. Unreliability, late delivery, lack of security — often even poor performance in large-scale systems can all be seen as deriving ultimately from unmanageable complexity. The primary status of complexity as* the *major cause of these other problems comes simply from the fact that being able to* understand *a system is a prerequisite for avoiding all of them, and of course it is this which complexity destroys."*

- Ben Moseley and Peter Marks, *Out of the Tar Pit*

goal: avoid accidental complexity by reducing/controlling shared state manipulation

don’t make concurrent programming harder than necessary!

Alternative concurrent programming models:

1. Actor model
2. Software Transactional Memory

1. Actor model
   - “actors” are processing primitives
     - do not share state
     - are *concurrent* & *independent*
     - interact via *asynchronous message passing*

e.g., Erlang

- created at Ericsson for telecom apps
- designed for concurrent, distributed, real-time systems
- “99.9999999 percent reliability (9 nines, or 31 ms. downtime a year!)”

- functional core
- creating actors (aka processes) is *cheap* (scales to millions of processes)
- essential architecture: client/server

```erlang
% basic pattern matching; note vars in uppercase
factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

% if expression: one guard must evaluate to true
max(A,B) -> if A < B -> B;
              true  -> A
            end.

% atoms (lowercase) and fixed-arity tuples
area({circle, R})       -> 3.1415 * R * R;
area({rectangle, L, W}) -> L * W;
area({square, L})       -> area({rectangle, L, L});
area(_)                 -> unknown.
```

```
> factorial(10).
3628800
> max(5, 10).
10
> area({rectangle, 5, 10}).
50
> area({triangle, 4, 5, 6}).
unknown
```

Creating processes:

```
Pid = spawn(Fun)
```

Sending messages (asynchronous):

```
Pid ! Message
```

Receiving messages (synchronous):

```
receive Pattern1 -> Expr1; ... end
```

Receiving with timeout:

```
receive ... after Millis -> Expr end
```

Server template:

```
loop() ->
    receive
        terminate -> done;
        Message   -> process(Message),
                     loop()
    end.
```

Server with “state”:

```
loop(State) ->
    receive
        terminate -> done;
        Message   -> loop(process(Message, State))
    end.
```

```
start() ->
    Pid = spawn(fun loop/0),    % create process
    Pid ! {self(), "hello!"},   % send it a message
    receive                     % block for a reply
        Msg -> io:format("Got ~s~n", [Msg])
    end,
    Pid ! terminate.            % tell it to shut down


loop() ->
    receive                     % block for a message
        {From, Msg} ->
            io:format("Got ~s~n", [Msg]),
            From ! lists:reverse(Msg),
            loop();             % re-run server loop
        terminate   ->
            io:format("Stopping~n")
    end.
```

```
> start().
Got hello!
Got !olleh
Stopping
```

```
consumer() ->
    % block for msg from producer
    receive
        terminate -> done;
        Val -> io:format("C: got ~w~n", [Val]),
               consumer()
    end.

producer(Val, Consumer) ->
    % send term msg or next value to consumer
    if Val =:= ?MAX_VAL ->
            Consumer ! terminate;
       true ->
            Consumer ! Val, % produce
            % loop to produce next value
            producer(Val + 1, Consumer)
    end.

start() ->
    C = spawn(fun consumer/0),
    % producer needs consumer pid & start value
    spawn(fun() -> producer(0, C) end).
```

```
consumer() ->
    % block for msg from producer
    receive
        terminate -> done;
        Val -> io:format("C: got ~w~n", [Val]),
               consumer()
    end.

producer(Val, Consumer) ->
    % send term msg or next value to consumer
    if Val =:= ?MAX_VAL ->
            Consumer ! terminate;
       true ->
            Consumer ! Val, % produce
            % loop to produce next value
            producer(Val + 1, Consumer)
    end.

start() ->
    C = spawn(fun consumer/0),
    % producer needs consumer pid & start value
    spawn(fun() -> producer(0, C) end).
```

MAX_VAL=10

```
C: got 0
C: got 1
C: got 2
C: got 3
C: got 4
C: got 5
C: got 6
C: got 7
C: got 8
C: got 9
```

- processes are automatically backed by “mailboxes” — by default unbounded
  - to simulate bounded buffer, must use messages to convey state & synch

```
producer(Val, Consumer, Ahead) ->
    if Val =:= ?MAX_VAL ->
            Consumer ! terminate;
       Ahead =:= ?MAX_AHEAD ->
            io:format("P: throttling!~n”), % force to wait for ack
            receive
                ack -> producer(Val, Consumer, Ahead - 1)
            end;
       true ->
            Consumer ! {self(), Val}, % produce
            io:format("P: ahead by ~w~n", [Ahead]),
            receive % try to get ack
               ack -> io:format("P: got ack),
                      producer(Val + 1, Consumer, Ahead)
            after % time out immediately if no ack in mailbox
                0 -> producer(Val + 1, Consumer, Ahead + 1)
            end
    end.

consumer() ->
    receive
        terminate -> done;
        {Producer, Val} -> io:format("C: got ~w~n", [Val]),
                           Producer ! ack, % send ack
                           consumer()
    end.
```

MAX_VAL=10, MAX_AHEAD=3

```
P: ahead by 0
C: got 0
P: ahead by 1
C: got 1
P: ahead by 2
P: got ack
P: ahead by 2
P: throttling!
C: got 2
P: ahead by 2
P: throttling!
C: got 3
P: ahead by 2
P: throttling!
```

```
C: got 4
C: got 5
P: ahead by 2
P: got ack
P: ahead by 2
P: throttling!
C: got 6
P: ahead by 2
P: throttling!
C: got 7
P: ahead by 2
C: got 8
C: got 9
```

```
producer(Val, Consumer, Ahead) ->
    if Val =:= ?MAX_VAL ->
            Consumer ! terminate;
       Ahead =:= ?MAX_AHEAD ->
            io:format("P: throttling!~n”), % force to wait for ack
            receive
                ack -> producer(Val, Consumer, Ahead - 1)
            end;
       true ->
            Consumer ! {self(), Val}, % produce
            io:format("P: ahead by ~w~n", [Ahead]),
            receive % try to get ack
               ack -> io:format("P: got ack),
                      producer(Val + 1, Consumer, Ahead)
            after % time out immediately if no ack in mailbox
                0 -> producer(Val + 1, Consumer, Ahead + 1)
            end
    end.
```

subtle issue: once producer hits cap, will never drop below ?

MAX_AHEAD-1

```
producer(Val, Consumer, Ahead) ->
    if Val =:= ?MAX_VAL ->
            Consumer ! terminate;
       Ahead =:= ?MAX_AHEAD ->
            io:format("P: throttling!~n”), % force to wait for ack
            receive
                ack -> producer(Val, Consumer, Ahead - 1)
            end;
       true ->
            io:format("P: ahead by ~w~n", [Ahead]),
            receive % process ack then loop
                ack -> io:format("P: got ack~n"),
                       producer(Val, Consumer, Ahead - 1)
            after % produce when timed out with no ack
                0 -> Consumer ! {self(), Val}, % produce
                     producer(Val + 1, Consumer, Ahead + 1)
            end
    end.
```

should process as many acks as possible before producing

MAX_VAL=10, MAX_AHEAD=3

```
P: ahead by 0
C: got 0
P: ahead by 1
P: got ack
P: ahead by 0
C: got 1
P: ahead by 1
P: ahead by 2
P: throttling!
C: got 2
P: ahead by 2
P: throttling!
C: got 3
C: got 4
P: ahead by 2
```

```
P: got ack
P: ahead by 1
P: got ack
P: ahead by 0
C: got 5
P: ahead by 1
P: ahead by 2
P: throttling!
C: got 6
P: ahead by 2
P: throttling!
C: got 7
P: ahead by 2
C: got 8
C: got 9
```

```
producer(Val, Consumer, Ahead) ->
    if Val =:= ?MAX_VAL ->
            Consumer ! terminate;
       Ahead =:= ?MAX_AHEAD ->
            io:format("P: throttling!~n”), % force to wait for ack
            receive
                ack -> producer(Val, Consumer, Ahead - 1)
            end;
       true ->
            io:format("P: ahead by ~w~n", [Ahead]),
            receive % process ack then loop
                ack -> io:format("P: got ack~n"),
                       producer(Val, Consumer, Ahead - 1)
            after % produce when timed out with no ack
                0 -> Consumer ! {self(), Val}, % produce
                     producer(Val + 1, Consumer, Ahead + 1)
            end
    end.
```

takeaway: Erlang doesn’t magically take care of synchronization issues!

dining philosophers in Erlang?

```erlang
%% “footman” server loop for distributing forks
loop(Forks) ->
    receive
        {Pid, {request, Fork}} ->
            case lists:member(Fork, Forks) of
                true ->
                    Pid ! {self(), granted},
                    loop(lists:delete(Fork, Forks));
                false ->
                    Pid ! {self(), unavailable},
                    loop(Forks)
            end;
        {Pid, {release, Fork}} ->
            Pid ! {self(), ok},
            loop([Fork|Forks]);
        {Pid, status} ->
            Pid ! {self(), Forks},
            loop(Forks);
        terminate ->
            ok
    end.


start(N) ->
    spawn(fun() -> loop(lists:seq(0,N-1)) end).
```

```
> Footman = forks:start(5).
<0.145.0>

> Footman ! {self(), status}.

> flush().
Shell got {<0.145.0>,[0,1,2,3,4]}

> Footman ! {self(), {request, 0}}.
> Footman ! {self(), status}.

> flush().
Shell got {<0.145.0>,granted} Shell got
{<0.145.0>,[1,2,3,4]}

> Footman ! {self(), {release, 0}}.
> Footman ! {self(), {request, 2}}.
> Footman ! {self(), {request, 2}}.
> Footman ! {self(), status}.

> flush().
Shell got {<0.145.0>,ok}
Shell got {<0.145.0>,granted}
Shell got {<0.145.0>,unavailable}
Shell got {<0.145.0>,[0,1,3,4]}
```

```erlang
%%% footman API; take care of acks

request(Pid, Fork) ->
    Pid ! {self(), {request, Fork}},
    receive
        {Pid, Msg} -> Msg
    end.

release(Pid, Fork) ->
    Pid ! {self(), {release, Fork}},
    receive
        {Pid, Msg} -> Msg
    end.
```

```
%% fork ids
leftFork(N) -> N.
rightFork(N) -> (N + 1) rem ?NUM_PHILOSOPHERS.

%% philosopher get-fork behavior: keep polling footman
getFork(Footman, Fork) ->
    case forks:request(Footman, Fork) of
        granted -> ok;
        unavailable -> io:format("Fork ~w unavailable~n", [Fork]),
                       timer:sleep(random:uniform(1000)),
                       getFork(Footman, Fork)
    end.

releaseFork(Footman, Fork) ->
    forks:release(Footman, Fork).
```

```
%% philosopher behavior
philosophize(_, _, 0) -> done;
philosophize(Id, Footman, NumMeals) ->
    getFork(Footman, leftFork(Id)),
    io:format("Philosopher ~w got fork ~w~n", [Id, leftFork(Id)]),
    getFork(Footman, rightFork(Id)),
    io:format("Philosopher ~w is eating!~n", [Id]),
    timer:sleep(random:uniform(1000)),
    releaseFork(Footman, leftFork(Id)),
    releaseFork(Footman, rightFork(Id)),
    philosophize(Id, Footman, NumMeals - 1).

start() ->
    Footman = forks:start(?NUM_PHILOSOPHERS),
    % spawn philosophers with unique ids & 1 footman; eat 500 meals
    [ spawn(fun() -> philosophize(N, Footman, 500) end)
      || N <- lists:seq(0, ?NUM_PHILOSOPHERS - 1)].
```

```
> philosophers:start().
Philosopher 0 got fork 0
Philosopher 1 got fork 1
Philosopher 2 got fork 2
Philosopher 3 got fork 3
Philosopher 4 got fork 4
Fork 1 unavailable
Fork 2 unavailable
Fork 3 unavailable
Fork 4 unavailable
Fork 0 unavailable
```

takeaway: Erlang doesn't magically take care of synchronization issues!

```
%% updated to restrict number of outstanding philosopher requests
loop(Forks, Phils) ->
    receive
        {Pid, {request, Fork}} ->
            % increment counter / add entry for requesting philosopher
            NextPhils = dict:update_counter(Pid, 1, Phils),
            % deny request if unavailable OR too many outstanding requests
            case lists:member(Fork, Forks) and (dict:size(NextPhils) < ?NUM_PHILS) of
                true  ->
                    Pid ! {self(), granted},
                    loop(lists:delete(Fork, Forks), NextPhils);
                false ->
                    Pid ! {self(), unavailable},
                    loop(Forks, Phils)
            end;
        {Pid, {release, Fork}} ->
            Pid ! {self(), ok},
            % remove dictionary entry on second release
            case (dict:fetch(Pid, Phils) =:= 1) of
                true  -> loop([Fork|Forks], dict:erase(Pid, Phils));
                false -> loop([Fork|Forks], dict:update_counter(Pid, -1, Phils))
            end;
        ...
    end.
```

```
Philosopher 0 got fork 0
Philosopher 1 got fork 1
Philosopher 2 got fork 2
Philosopher 3 got fork 3
Fork 4 unavailable
Fork 1 unavailable
Fork 2 unavailable
Fork 3 unavailable
Philosopher 3 is eating!
Philosopher 2 is eating!
Fork 2 unavailable
Fork 1 unavailable
Fork 4 unavailable
Fork 3 unavailable
Fork 3 unavailable
Philosopher 4 got fork 4
Fork 1 unavailable
Fork 2 unavailable
Fork 0 unavailable
Philosopher 2 got fork 2
Philosopher 2 is eating!
...
```

Process synchronization is still an issue!

- But is now our *primary focus* (i.e., less *accidental* complexity!)
- Typically reuse well known *patterns*
  - e.g., *ring/star* configurations

Messages may be big — but no other way of sharing data!

- Runtime can optimize this using shared memory and other techniques due to immutability

We've eliminated shared state issues!

Huge boon to reasoning, composability, and robustness

- actors are independent — if down or unresponsive, can route around it

Also, makes deploying on distributed hardware transparent

Projects in Erlang:

- Facebook Chat
- RabbitMQ messaging framework
- Amazon SimpleDB, Apache CouchDB
- lots of telephony and real-time (e.g., routing, VOIP) services

for more information:

- http://www.erlang.org/
- http://learnyousomeerlang.com/

2. Software Transactional Memory (STM)

- supports shared memory
- but *all changes* are vetted by runtime

STM guarantees ACID properties:

- **A**tomicity
- **C**onsistency
- **I**solation

Atomicity:

- all requested changes take place (commit), or none at all (rollback)

Consistency:

- updates always leave data in a *valid state*
- i.e., allow validation hooks

Isolation:

- no transaction sees intermediate effects of other transactions

e.g., Clojure

- “invented” by Rich Hickey
- a (mostly functional) Lisp dialect
- primarily targets JVM

synchronization is *built into* the platform

based on a re-examination of *state vs. identity*

Tenet: most languages (notably, OOPLs) simplify but complicate *identity*

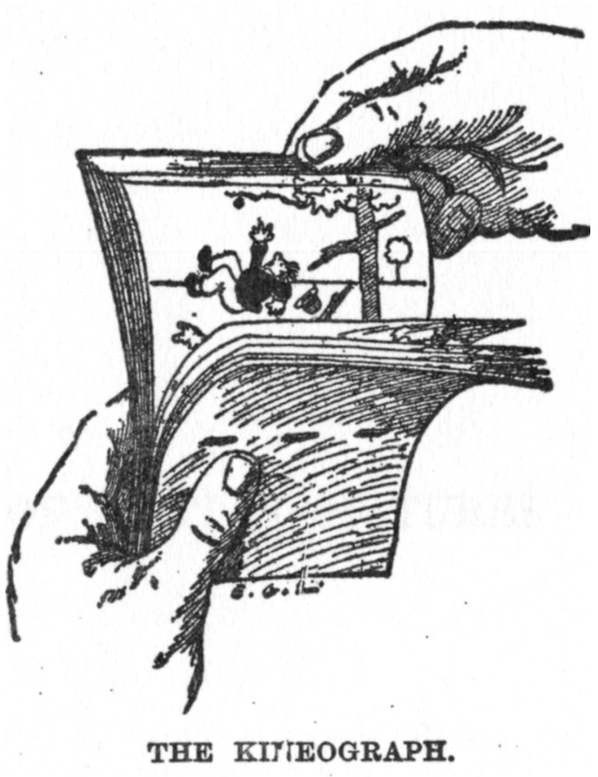
- *identity* is conflated with *state*
- an object's state (attributes) can change, and it's still considered *the same object*
  - e.g., pointer based equality

Ramifications:

- threads can concurrently change the state of the same object
- objects that are entirely identical (state-wise) are considered different
  - requires comparators, .equals, etc.

Alternate view: objects perpetually advance through separate, *instantaneous states*

- *state cannot change!*
- but we can use names (i.e., *references*) to refer to the *most recent* state

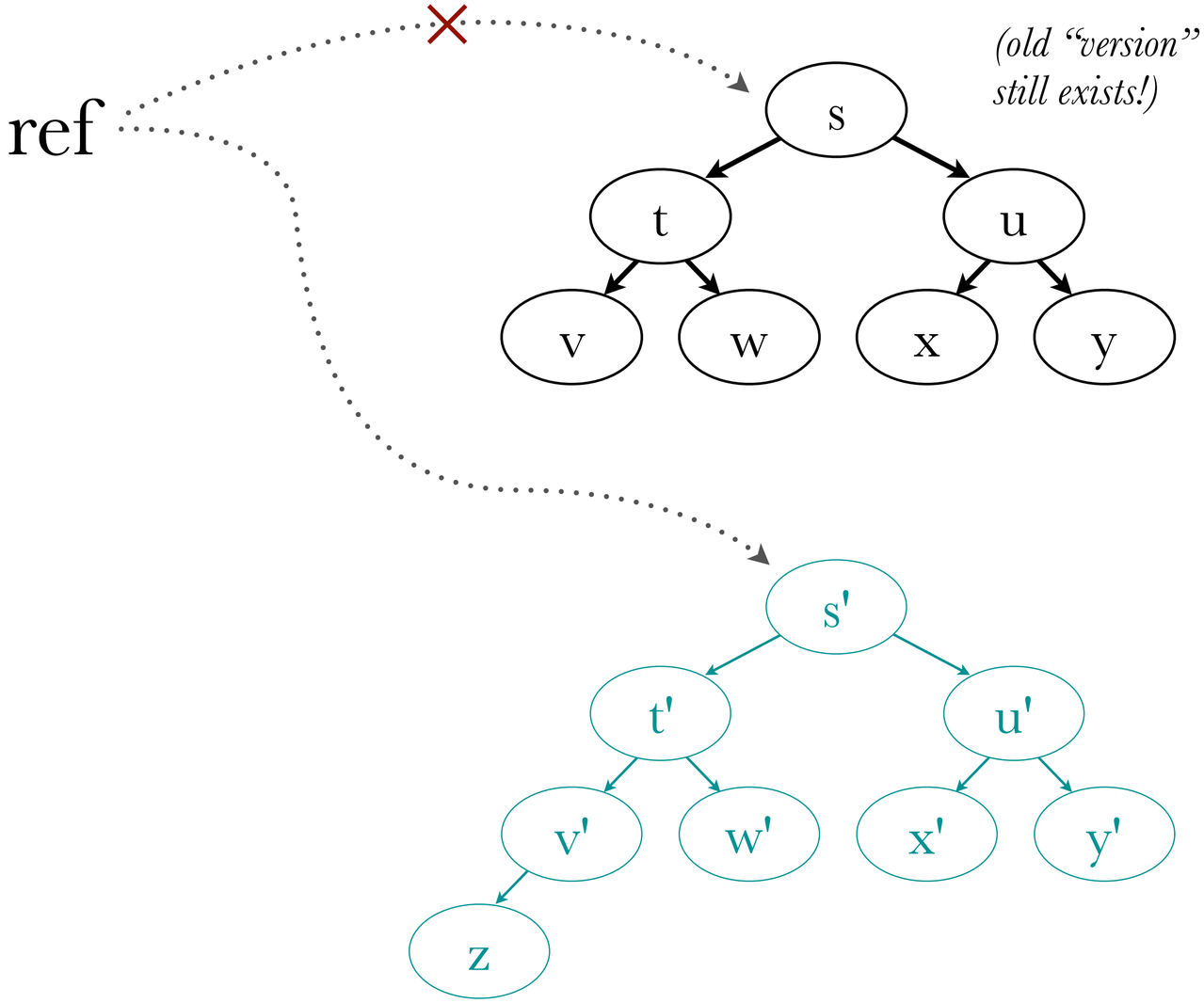


In Clojure, all values (state) are *immutable*

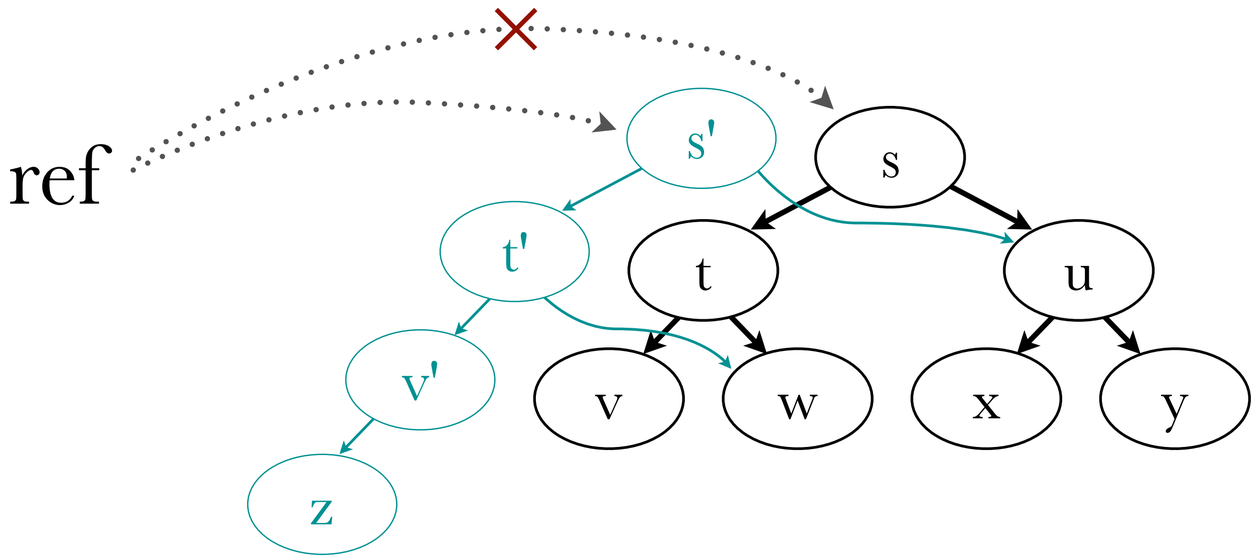
… but we can point a given *reference* at different states

to “update” a data structure:

1. access current value via reference
2. use it to create a new value
3. modify reference to refer to new value


ref
(old "version"
still exists!)
s
t
u
v
w
x
y
s'
t'
u'
v'
w'
x'
y'
z

problem: very inefficient for large data structures



in practice, share structure with old version

"persistent" data structures

- allow for structural sharing
  - ok because they are *immutable*
- allow *multiple versions* of a given data structure to be kept around

Multiversion Concurrency Control (**MVCC**)

- track versions of all data in history
- support “point-in-time” view of all data

Value vs. Reference dichotomy is crucial

- *immutable values* let us use data without concern that it will change under us
- *references* let us easily coordinate “changes”

important: how can we alter references?

- if arbitrarily, still have synch issue
- Clojure has multiple types of references, with different "change" semantics

Clojure reference types:

- vars
- atoms
- refs
- agents

vars are classic “variables”

- bound to *root values*, shared by all threads
- bad style to change at runtime
  - i.e., treat bound values as constants

```
;;; vars

(def x 10)
(inc x) ; => 11
x ; => 10 (unchanged)

(def acc {:name "checking" :balance 1000})

(defstruct account :name :balance)
(def acc2 (struct account "savings" 2000))

(= acc2 {:name "savings" :balance 2000}) ; => true

(def acc3 (assoc acc2 :name "business"))

acc3 ; => {:name "business" :balance 2000}
acc2 ; => {:name "savings" :balance 2000} (unchanged)
```

atoms support *isolated*, *atomic* updates

- provide with a function to compute new value from old value
  - atom is updated in mutex

```
;;; atoms

(def count (atom 0))

(deref count) ; => 0

@count ; => 0 (‘@’ is shortcut for deref)

(swap! count inc)

@count ; => 1

(reset! count 0)

@count ; => 0
```

`swap` runs function on atom's *current value*

- if another thread changes the atom before I write my update, retry!

E.g., concurrent increments:

```python
N_THREADS = int(sys.argv[1])
N_INCS    = int(sys.argv[2])

count = 0
threads = []

def do_incs(n):
    global count
    for _ in range(n):
        count = count + 1

for _ in range(N_THREADS):
    threads.append(Thread(target=do_incs,
                          args=[N_INCS]))

for t in threads: t.start()
for t in threads: t.join()

print(count)
```

```
python inc.py 10 100        =>    1000

python inc.py 10 1000       =>   10000

python inc.py 10 10000      =>  100000

python inc.py 10 100000     =>  949034

python inc.py 10 1000000    => 3300032
```

```clojure
(def count (atom 0))

(defn do-incs [n]
  (dotimes [_ n]
    (swap! count inc)))

(defn run [nthreads nincs]
  (reset! count 0)
  (let [pool
        (Executors/newFixedThreadPool
         nthreads)]
    (dotimes [_ nthreads]
      (.execute pool #(do-incs nincs)))
    (.shutdown pool)
    (.awaitTermination pool
                       600
                       TimeUnit/SECONDS)
    (println @count)))
```

```clojure
(run 10 100)       =>      1000

(run 10 1000)      =>     10000

(run 10 10000)     =>    100000

(run 10 100000)    =>   1000000

(run 10 1000000)   => 10000000
```

refs support *coordinated* updates

- updates can only take place in *transactions*
  - demarcated with `dosync` keyword
  - within a transaction, we automatically get atomicity/isolation

```
;;; refs

(def a (ref 10))
(def b (ref 20))

(defn swap [ref1 ref2]
  (dosync ; start transaction
   (let [val1 @ref1
         val2 @ref2]
     (ref-set ref1 val2)
     (ref-set ref2 val1))))

(swap a b) ; @a = 20, @b = 10

(dosync (alter a inc)) ; @a = 21
```

E.g., concurrent swaps:

```
public class ConcurSwap {
    int numLists;
    int numItemsPerList;
    int numThreads;
    int numIterations;

    private List<List<Integer>> sharedData;
    private ExecutorService threadPool;

    public ConcurSwap (int nLists, int nItems, int nThreads, int nIters) {
        numLists = nLists;
        numItemsPerList = nItems;
        numThreads = nThreads;
        numIterations = nIters;
        sharedData = new ArrayList<List<Integer>>(numLists);
        for (int i=0, val=0; i<numLists; i++) {
            List<Integer> l = Collections.synchronizedList(
                                    new ArrayList<Integer>(numItemsPerList));
            for (int j=0; j<numItemsPerList; j++) {
                l.add(val++);
            }
            sharedData.add(l);
        }
        threadPool = Executors.newFixedThreadPool(numThreads);
    }
```

```
class Swapper implements Runnable {
    public void run () {
        Random randGen = new Random();
        for (int i=0; i<numIterations; i++) {
            int idx1 = randGen.nextInt(numItemsPerList),
                idx2 = randGen.nextInt(numItemsPerList);
            List<Integer> lst1 = sharedData.get(randGen.nextInt(numLists)),
                lst2 = sharedData.get(randGen.nextInt(numLists));

            int tmpVal = lst1.get(idx1);
            lst1.set(idx1, lst2.get(idx2));
            lst2.set(idx2, tmpVal);
        }
    }
}


public void addSwapper () {
    threadPool.execute(new Swapper());
}

public void await () {
    try {
        threadPool.shutdown();
        threadPool.awaitTermination(60, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

```
    public void report () {
        Set<Integer> uniquer = new HashSet<Integer>();
        for (List<Integer> l : sharedData) {
            System.out.println(l.toString());
            uniquer.addAll(l);
        }
        System.out.printf("Unique items: %d\n", uniquer.size());
    }


    public static void main (String[] args) {
        int nLists   = Integer.parseInt(args[0]),
            nItems   = Integer.parseInt(args[1]),
            nThreads = Integer.parseInt(args[2]),
            nIters   = Integer.parseInt(args[3]);
        ConcurSwap syncTest = new ConcurSwap(nLists, nItems, nThreads, nIters);
        syncTest.report();
        for (int i=0; i<nThreads; i++) {
            syncTest.addSwapper();
        }
        syncTest.await();
        syncTest.report();
    }
} // end ConcurSwap
```

```
$ java ConcurSwap 5 10 1 10000
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
Unique items: 50
[40, 27, 48, 41, 19, 7, 17, 9, 1, 49]
[43, 4, 3, 29, 39, 2, 0, 5, 12, 47]
[26, 35, 6, 24, 8, 30, 28, 33, 14, 38]
[21, 37, 15, 36, 22, 31, 34, 13, 20, 32]
[45, 25, 44, 46, 11, 18, 42, 16, 10, 23]
Unique items: 50
```

```
$ java ConcurSwap 5 10 5 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
Unique items: 50
[0, 8, 30, 3, 25, 23, 7, 2, 16, 43]
[36, 11, 37, 33, 14, 32, 4, 17, 38, 13]
[42, 21, 18, 47, 19, 27, 26, 12, 28, 10]
[29, 1, 5, 15, 45, 35, 24, 6, 22, 31]
[34, 41, 9, 48, 44, 20, 49, 40, 39, 46]
Unique items: 50
```

```
$ java ConcurSwap 5 10 5 100
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
Unique items: 50
[21, 12, 10, 18, 49, 0, 30, 19, 4, 26]
[7, 46, 47, 28, 27, 38, 31, 41, 29, 20]
[42, 32, 34, 17, 22, 9, 15, 13, 32, 25]
[35, 44, 24, 8, 44, 45, 23, 37, 18, 43]
[34, 5, 39, 40, 1, 2, 14, 16, 48, 32]
Unique items: 45
```

```
$ java ConcurSwap 5 10 10 1000
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
Unique items: 50
[38, 29, 14, 19, 23, 38, 23, 29, 29, 23]
[19, 23, 29, 34, 38, 14, 19, 34, 0, 29]
[23, 14, 23, 29, 21, 29, 29, 19, 23, 19]
[19, 29, 29, 38, 29, 29, 19, 29, 21, 29]
[29, 38, 19, 38, 29, 29, 34, 29, 0, 34]
Unique items: 8
```

```
$ java ConcurSwap 10 10 10 10000
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
[50, 51, 52, 53, 54, 55, 56, 57, 58, 59]
[60, 61, 62, 63, 64, 65, 66, 67, 68, 69]
[70, 71, 72, 73, 74, 75, 76, 77, 78, 79]
[80, 81, 82, 83, 84, 85, 86, 87, 88, 89]
[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
Unique items: 100
[97, 97, 82, 94, 72, 72, 72, 97, 72, 82]
[36, 36, 94, 97, 36, 97, 97, 94, 72, 72]
[79, 72, 97, 97, 72, 36, 94, 94, 94, 94]
[72, 36, 72, 72, 72, 72, 36, 72, 97, 82]
[79, 94, 94, 94, 36, 82, 97, 36, 97, 36]
[82, 97, 94, 97, 94, 72, 72, 72, 72, 72]
[94, 72, 94, 72, 72, 94, 36, 94, 94, 36]
[97, 72, 94, 72, 72, 94, 94, 94, 72, 94]
[36, 72, 72, 97, 72, 97, 36, 72, 94, 72]
[97, 94, 94, 72, 97, 72, 82, 72, 94, 94]
Unique items: 6
```

```
import java.util.concurrent.locks.*;

...

    private List<Lock> locks;

    public ConcurSwapSync (int nLists, int nItems, int nThreads, int nIters) {
        sharedData = new ArrayList<List<Integer>>(numLists);
        locks = new ArrayList<Lock>(numLists);
        for (int i=0, val=0; i<numLists; i++) {
            List<Integer> l = Collections.synchronizedList(
                                new ArrayList<Integer>(numItemsPerList));
            ...
            sharedData.add(l);
            locks.add(new ReentrantLock());
            threadPool = Executors.newFixedThreadPool(numThreads);
        }
    }
```

```
class Swapper implements Runnable {
     public void run () {
         for (int i=0; i<numIterations; i++) {
             int idx1 = randGen.nextInt(numItemsPerList),
                 idx2 = randGen.nextInt(numItemsPerList);
             int lidx1 = randGen.nextInt(numLists),
                 lidx2 = randGen.nextInt(numLists);
             List<Integer> lst1 = sharedData.get(lidx1),
                           lst2 = sharedData.get(lidx2);
             Lock lock1 = locks.get(lidx1),
                  lock2 = locks.get(lidx2);

             lock1.lock();
             lock2.lock();
             try {
                 int tmpVal = lst1.get(idx1);
                 lst1.set(idx1, lst2.get(idx2));
                 lst2.set(idx2, tmpVal);
             } finally {
                 lock1.unlock();
                 lock2.unlock();
             }
         }
     }
 }
```

```
$ java ConcurSwap2 5 10 10 1000
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
Unique items: 50
(deadlock!)
```

```clojure
;; create refs to nvecs vectors, each with nitems items
(defn make-vecs [nvecs nitems]
  (map (comp ref vec)
       (partition nitems (range (* nvecs nitems)))))

(defn rand-swap [vec-refs nvecs nitems]
  (let [v1ref (nth vec-refs (rand-int nvecs))
        idx1 (rand-int nitems)
        v2ref (nth vec-refs (rand-int nvecs))
        idx2 (rand-int nitems)]
    (dosync  ; do the swap in a transaction
     (let [tmp (nth @v1ref idx1)]
       (alter v1ref assoc idx1 (nth @v2ref idx2))
       (alter v2ref assoc idx2 tmp)))))
```

```clojure
(defn report [vec-refs]
  (let [vecs (map deref vec-refs)]
    (pprint vecs)
    (println "Unique items: "
             (count (distinct (apply concat vecs))))))

(defn run [nvecs nitems nthreads niters]
  (let [vec-refs (make-vecs nvecs nitems)]
    (report vec-refs)
    (let [pool (Executors/newFixedThreadPool nthreads)]
      (dotimes [_ nthreads]
        (.execute pool #(dotimes [_ niters]
                          (rand-swap vec-refs nvecs nitems))))
      (.shutdown pool)
      (.awaitTermination pool 60 TimeUnit/SECONDS))
    (report vec-refs)))
```

```
(run 5 10 5 10)
([0 1 2 3 4 5 6 7 8 9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49])
Unique items:  50
([6 30 32 28 33 12 46 7 8 11]
 [4 21 47 17 14 0 3 5 49 20]
 [19 43 22 10 24 25 26 40 23 29]
 [38 31 13 27 16 2 36 34 44 39]
 [35 37 1 41 15 48 45 9 42 18])
Unique items:  50
```

```
(run 10 10 10 100000)
([0 1 2 3 4 5 6 7 8 9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]
 [50 51 52 53 54 55 56 57 58 59]
 [60 61 62 63 64 65 66 67 68 69]
 [70 71 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89]
 [90 91 92 93 94 95 96 97 98 99])
Unique items:  100
([57 5 16 83 37 22 1 23 99 24]
 [19 49 78 20 27 94 62 48 79 28]
 [40 39 91 86 7 30 93 64 13 14]
 [15 56 0 65 46 90 47 44 58 66]
 [35 9 80 97 71 69 98 88 61 2]
 [50 55 41 38 82 87 68 21 81 54]
 [33 63 92 75 18 45 70 42 36 95]
 [31 4 6 26 89 25 52 96 51 77]
 [43 84 17 11 72 8 10 85 3 73]
 [67 32 59 29 60 53 12 34 76 74])
Unique items:  100
```

important: transaction is run *optimistically*

- refs are all updated at single *commit-point*
- if another transaction changes *any of the refs* I'm altering before I commit, rerun transaction *from the beginning*!
  - alternative: `commute` (vs. `alter`)
    — only require latest ref value at commit time (not during transaction)

agents support *asynchronous* updates

- update functions run in separate thread
- at most one action being run at any time
  - i.e., updates (aka *actions*) are queued

```
;;; agents

(def bond (agent 007))

@bond ; => 7

(send bond inc)

;; a short while later...
@bond ; => 8

(apply pcalls ; run list of functions in parallel
       (repeat 10  ; create ten functions
               #(dotimes [_ 1000] ; that each send 1000
                  (send bond inc))))) ; inc's to agent

;; a short while later...
@bond ; => 10008
```

Demo: “cashflow” & “baboons”

Benefits of STM:

- automatic support for mutex/isolation
- optimistic transactions maximize concurrency
- framework helps guarantee freedom from race conditions!

Clojure-specific benefits:

- modifications to refs *must happen* within transactions (i.e., not advisory)
- persistent data structures allow for “snapshot” MVCC (vs. logging in other implementations)

Drawbacks:

- transaction restarts = overhead
  - performance is not transparent
    - compared to locking?
- MVCC = overhead (need a lot of GC)
- snapshot isolation → *write skew*

Write skew scenario:

- Accounts X, Y with total min balance M
  - Thread A debits X, checks $X + Y \leq M$
  - Thread B debits Y, checks $X + Y \leq M$
- Only conflicting *writes* require rollback!
  - A may read old version of Y (and B of X)

Clojure “fix” for write-skew:

- can pretend to update reference:
  `(ref-set ref @ref)`
- or use `(ensure ref)` — requires rollback if ref has been changed at commit point

§ Summary

Chips aren’t getting any faster, but we are getting more processing cores

— we need a scalable way of writing concurrent programs

Mutable, shared memory and locks are hard to reason about and add *unnecessary complexity* to programs (especially in concurrent settings)

Two alternative models:

- Actor model: no shared state, ever; communicate via messages
- STM: transactional support for state transactions

Implementations:

- Actor model: Erlang, Scala, node.js
- STM: Clojure, Haskell, Python, etc.

No silver bullet!

- Actor model: still need to worry about synchronization, deadlock still possible
- STM: performance under scrutiny

What would *you* use?

Bibliography:

- Harris, T., Marlow, S., & Peyton-Jones, S. *Composable memory transactions*. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '05).

- Peyton-Jones, S. *Beautiful concurrency*. Beautiful Code. 2007.

- Moseley, B & Marks, P. *Out of the tar pit*. 2006.

- Fraser, K., & Harris, T. *Concurrent programming without locks*. ACM Transactions on Computer Systems, 25(2), 5. 2007.

- Hansen, P. *Java's insecure parallelism*. SIGPLAN notices. 1999.

- Hickey, R. *Are we there yet?* JVM Languages Summit presentation. 2009.