Concurrency Models

CS 450 : Operating Systems

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.

"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*

Outline

- Status quo & Related issues
- Complexity & Causes
- Goals
- Concurrency models: Actor & STM

"Mutual-exclusion locks are one of the most widely used and fundamental abstractions for synchronization ... due to their apparently simple programming model and the availability of implementations which are efficient and scalable. 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. To avoid mistakes, this favors the development of simple locking strategies ...*
- *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*

Most common approach to concurrency:

- shared, mutable memory
- lock-based synchronization
  (e.g., semaphores, mutexes)

Position Sep 2011	Position Sep 2010	Delta in Position	Programming Language	Ratings Sep 2011	Delta Sep 2010	Status
1	1	=	Java	18.761%	+0.85%	A
2	2	=	C	18.002%	+0.86%	A
3	3	=	C++	8.849%	-0.96%	A
4	6	↑↑	C#	6.819%	+1.80%	A
5	4	↓	PHP	6.596%	-1.77%	A
6	8	↑↑	Objective-C	6.158%	+2.79%	A
7	5	↓↓	(Visual) Basic	4.420%	-1.38%	A
8	7	↓	Python	4.000%	-0.58%	A
9	9	=	Perl	2.472%	+0.03%	A
10	11	↑	JavaScript	1.469%	-0.20%	A
11	10	↓	Ruby	1.434%	-0.47%	A
12	12	=	Delphi/Object Pascal	1.313%	-0.27%	A
13	24	↑↑↑↑↑↑↑↑↑↑	Lua	1.154%	+0.60%	A
14	13	↓	Lisp	1.043%	-0.04%	A
15	15	=	Transact-SQL	0.860%	+0.09%	A
16	14	↓↓	Pascal	0.845%	+0.06%	A-
17	20	↑↑↑	PL/SQL	0.720%	+0.08%	A--
18	19	↑	Ada	0.682%	+0.01%	B
19	17	↓↓	RPG (OS/400)	0.666%	-0.05%	B
20	30	↑↑↑↑↑↑↑↑↑↑	D	0.609%	+0.20%	B

http://www.tiobe.com

TIOBE language popularity chart

most popular paradigms:
procedural & object-oriented

OO = data encapsulation via methods

i.e., tight coupling of *state* & *behavior*

"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 program", 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*

"This problem (that a test in one state *tells you nothing at all about the system in a different* state*) is a direct parallel to one of the fundamental problems with testing … namely 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."*

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

"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, for in this case, we can no longer even be assured of result consistency when repeating tests on a system — even if we somehow ensure a consistent starting state. 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*

OOP $\Rightarrow$ “big mutable balls”

OOP does not scale well with large amounts of concurrency (& explicit locking)!

Java implements *implicit locks* (monitors) for single object read/write.

(i.e., Java synchronized methods/blocks)

"The author examines the synchronization features of Java and finds that they are insecure variants of his earliest ideas in parallel programming published in 1972–73. The claim that Java supports monitors is shown to be false. The author concludes that Java ignores the last twenty-five years of research in parallel programming languages."

- Per Brinch Hansen, *Java's Insecure Parallelism*

OOP simplifies (but complicates) *identity*

- *identity* is disconnected from *state*
- an object's state (attributes) can change, but it's still considered *the same object*

Ramifications:

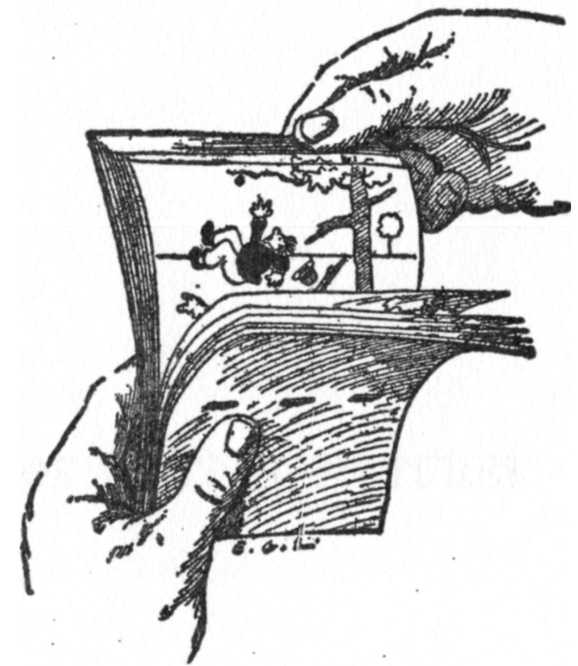
- threads can concurrently change the state of the same object
- objects that are entirely identical (state-wise) are considered different
  - requires comparators, .equals, etc.

“You cannot step twice into the same river; for other waters and yet others go ever flowing on.”

- Heracltus

In reality, objects are perpetually advancing through separate, *instantaneous states*

- we conveniently use names (i.e., *references*) to refer to the *most recent* state



THE KINEOGRAPH.

You can't modify *state*!

... but you can use the *same name*
to refer to different states

Another issue: composability

"For example, 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 issue!

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

object identity, lack of composability …

unnecessary, or *accidental* complexity

"Complexity is the *root cause of the vast majority of problems with software 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*

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

- Alfred North Whitehouse

goal: avoid accidental complexity

— don’t make concurrent programming harder than it needs to be!

Concurrency models:

1. Actor model
2. Software Transactional Memory

1. Actor model

- no shared state, ever
- actors are *concurrent* & *independent*
- actors only interact through *message passing*

excellent for **distributed systems**

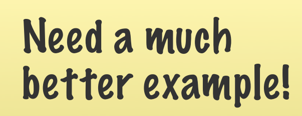
— doesn't matter if actors are local or not

e.g., Erlang

- functional core
- messages are *asynchronous* (send & pray)
- creating actors is *cheap* (scales to millions of processes)

```erlang
go() ->
    Pid2 = spawn(loop),
    Pid2 ! {self(), hello},
    receive
        {Pid2, Msg} ->
            io:format("P1 ~w~n", [Msg])
    end,
    Pid2 ! stop.

loop() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            loop();
        stop ->
            true
    end.
```

```
P1 hello
stop
```


Need a much
better example!

Issues:

- Coordination / synchronization is hard
- Messages may be big — no other way of sharing data
- Big data processing is potentially inefficient

2. Software Transactional Memory (STM)

- support shared memory
- access shared memory in *transactions*

STM guarantees ACI~~D~~ 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

(mostly) functional language

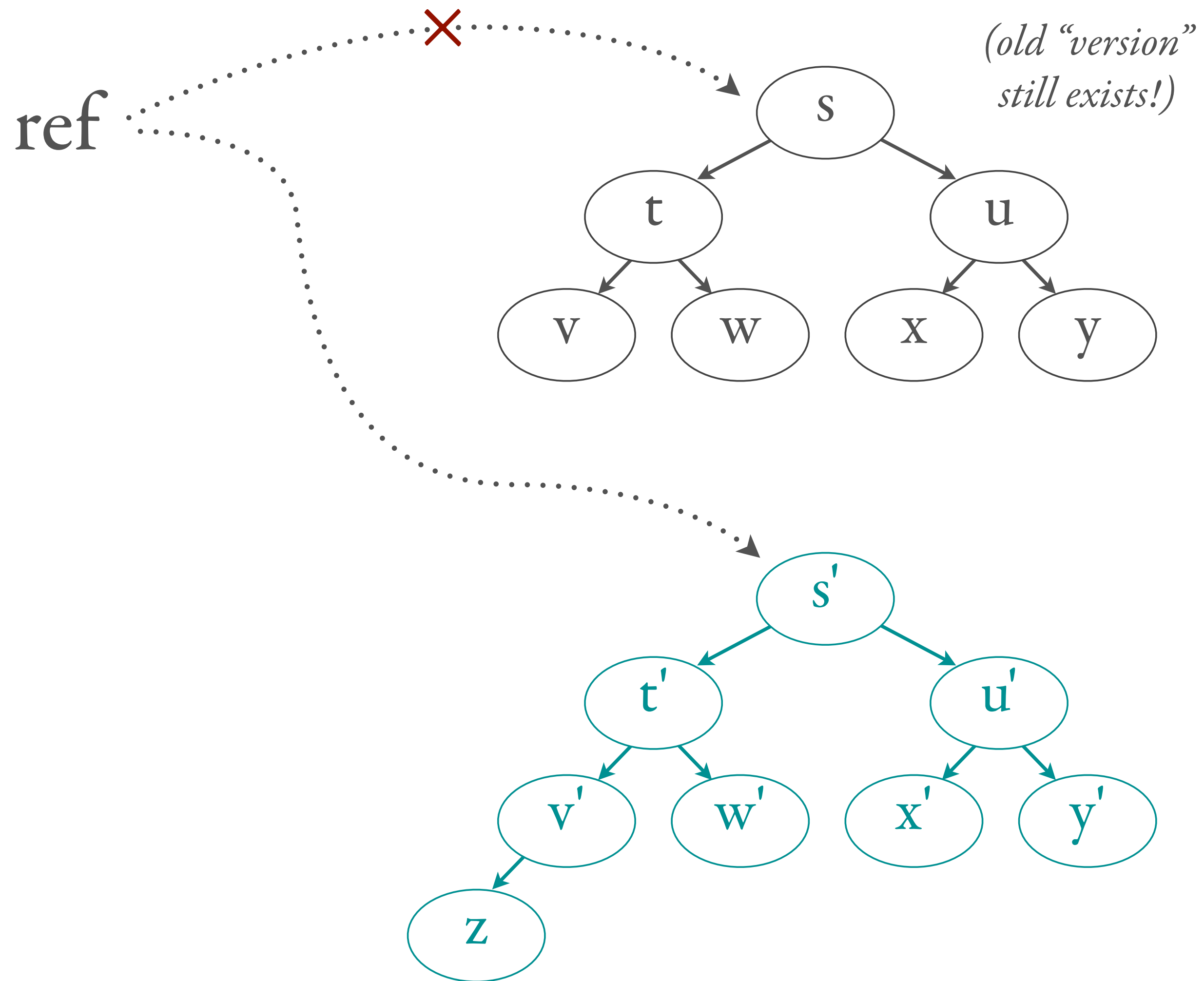
important premise: *all data* is ***immutable***

values cannot be changed in place

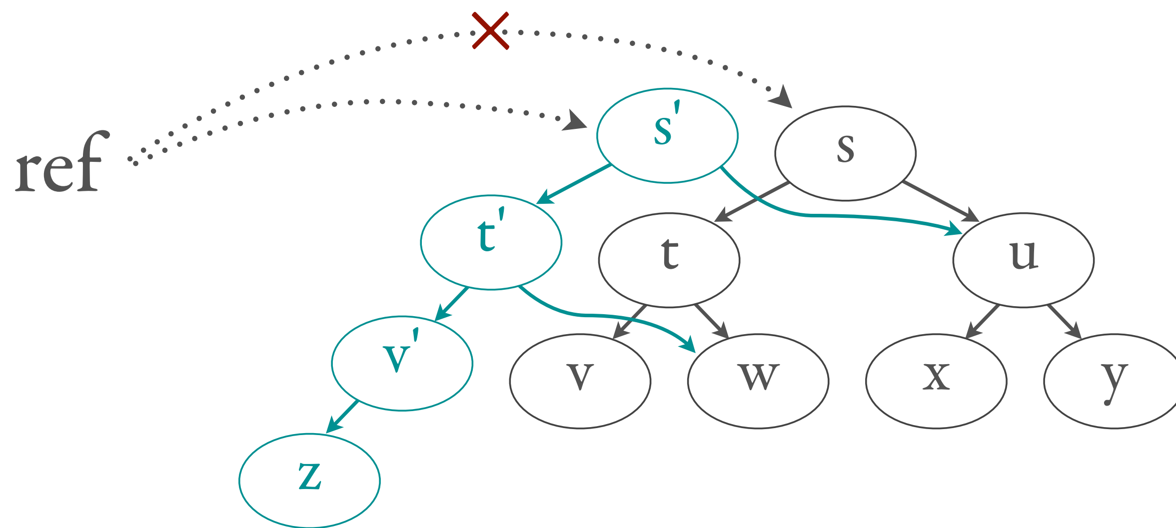
... but *references* can be!

to “change” a data structure:

1. access current value via reference
2. use it to create a new value
3. modify reference (in transaction) 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
  - precisely because they are *immutable*
- require garbage collection
- 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 allow us to coordinate “changes”
- MVCC gives us ability to look at a *snapshot* of the world

important: how/when can we alter references?

- if arbitrarily, still have synchronization issue
- at least, need to guarantee atomicity
- better: transactional support

Clojure reference types:

- vars
- atoms
- agents
- refs

vars provide for global *bindings*

- *root* bindings share by threads — very poor style to change these during runtime
- *dynamic* thread-local bindings

```clojure
;; global/root binding for var user (map)
(def user {:name "Michael"})

;; dereference directly using name of var
user ; => {:name "Michael"}

;; assoc function “changes” a map
(assoc user :alias "C5Pr0f") ; => {:name "Michael", :alias "C5Pr0f"}

user ; => {:name "Michael"}

;; root binding for var get-name (function)
(defn get-name []
  (:name user))

(get-name) ; => "Michael"

;; binding establishes new thread local binding
(binding [user {:name "Jane"}]
  (get-name)) ; => "Jane"

(get-name) ; => "Michael"
```

atoms support *atomic* changes

- synchronous – when change operation returns, change has occurred
- no coordinated updates to multiple atoms

```clojure
(def current-weapon (atom {:name "RPG" :damage 25}))

;; deref obtains value from reference
(deref current-weapon) ; => {:name "RPG", :damage 25}

;; @ is a shortcut for deref
@current-weapon ; => {:name "RPG", :damage 25}

;; reset! changes the current value of the atom
(reset! current-weapon {:name "Shotgun" :damage 5})

@current-weapon ; => {:name "Shotgun", :damage 5}

;; swap! applies the given function (and args) to the
;; current atom value to get its new value
(swap! current-weapon assoc :damage 30)

@current-weapon ; => {:name "Shotgun", :damage 30}
```

function given to swap! is run on *current value*

- run *optimistically*
- another thread may have updated the reference by the time we are done ...
  - in which case, *retry*!

important: update function *should not have* ***side-effects*** – must assume it may be called multiple times

refs support *coordinated* updates

- changes to refs occur in transactions
- in a transaction, we start with a consistent snapshot of the world (courtesy MVCC)

```clojure
(def game-state (ref {:points 0 :won false}))

@game-state ; => {:points 0, :won false}

(defn add-points [n]
  (dosync ; everything in here is a transaction
    (when (not (:won @game-state))
      (let [newpts (+ n (:points @game-state))]
        ;; alter calls the supplied function with the current
        ;; value of the ref to compute the new value
        (alter game-state assoc :points newpts)
        (if (>= newpts 100)
          (alter game-state assoc :won true))))))

(add-points 10)

@game-state ; => {:points 10, :won false}

(add-points 90)

@game-state ; => {:points 100, :won true}
```

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, *retry*

sometimes, this is too strict …

- maybe the transaction doesn't care if the value has changed
  - so long as the altering function is given the most recent version

```clojure
(defn add-achievement
  ;; commute instead of alter -- allows more concurrency
  ;; by using the most recent value of refs at commit
  ([k v] (dosync (commute game-state merge {k v})))
  ([a] (dosync (commute game-state
                        (partial merge-with concat)
                        {:achievements [a]}))))

(add-achievement :killed-boss "You killed the boss!")

@game-state ; => {:killed-boss "You killed the boss!", :points 100, etc.}

(add-achievement "Level 1 beat!")
(add-achievement "Level 2 beat!")

@game-state ; => {:achievements ("Level 1 beat!" "Level 2 beat!"), etc.}
```

slide on write-skew & ensure

agents support *asynchronous* updates

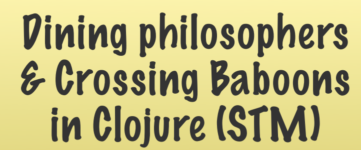
- change functions run in separate thread — called agent *actions*
- at most one action being run on an agent at any time

```clojure
(def player (agent {:pos [0 0] :health 100}))

@player ; {:pos [0 0], :health 100}

;; agent action -- will be applied to the current agent state (first arg)
(defn move-player [player [dx dy]]
  (let [[nx ny] (map + [dx dy] (:pos player))
        damage (do-battle [nx ny])
        nhealth (- (:health player) damage)]
    (if (<= nhealth 0)
      (dosync (alter game-state assoc :game-over true)))
    ;; return the new agent state
    (assoc player :pos [nx ny] :health nhealth)))

(defn do-battle [[x y]] 5) ; placeholder for battle logic


(send player move-player [1 1]) ; => action dispatch (immediate return)

;; a little later ...
@player ; => {:pos [1 1], :health 95}

;; dispatch multiple requests to agent "mailbox"
(dotimes [n 19] (send player move-player [1 1]))

;; a little later ...
@player ; => {:pos [20, 20], :health 0}
@game-state ; => {:game-over true, etc.}
```

```clojure
;; autopilot is an agent action that sends multiple move-player actions
;; ... need to access the original agent reference to send it actions
(defn autopilot [player n-moves [dx dy]]
  (when (> n-moves 0)
    ;; *agent* is a thread-local binding that gives us this ref
    (send *agent* move-player [dx dy]) ; queue up a move-player action
    (send *agent* autopilot (dec n-moves) [dx dy])) ; autopilot (not recursive!)
  player) ; return the current value (not yet moved)



;; init value
(def player (agent {:pos [0 0] :health 100}))

(send player autopilot 5 [1 1])

;; a little later ...
@player ; => {:pos [5, 5], :health 75}
```

	function	*arguments*
atoms	`reset!`	new value
	`swap!`	function to apply to old value
refs	`alter`	function to apply to old value
	`commute`	function to apply to old value
agents	`send`	function to apply to old value

Demo



Benefits of STM:

- automatic support for ACI (DB) properties
- optimistic transactions maximize concurrency
- framework can guarantee *freedom from race conditions* (deadlock, livelock, etc.)

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 transparency?
- MVCC = overhead (need a lot of GC)
- snapshot isolation → *write skew*