Concurrent Programming Paradigms

CS 450: Operating Systems

Michael Lee <lee@iit.edu>

ILLINOIS TECH | College of Computing

Agenda

- Traditional concurrent programming
  - Shared-memory paradigm
- Alternative concurrent programming paradigms
  - Software transactional memory (STM)
  - Message-passing
  - Event-driven

§ Traditional concurrent programming

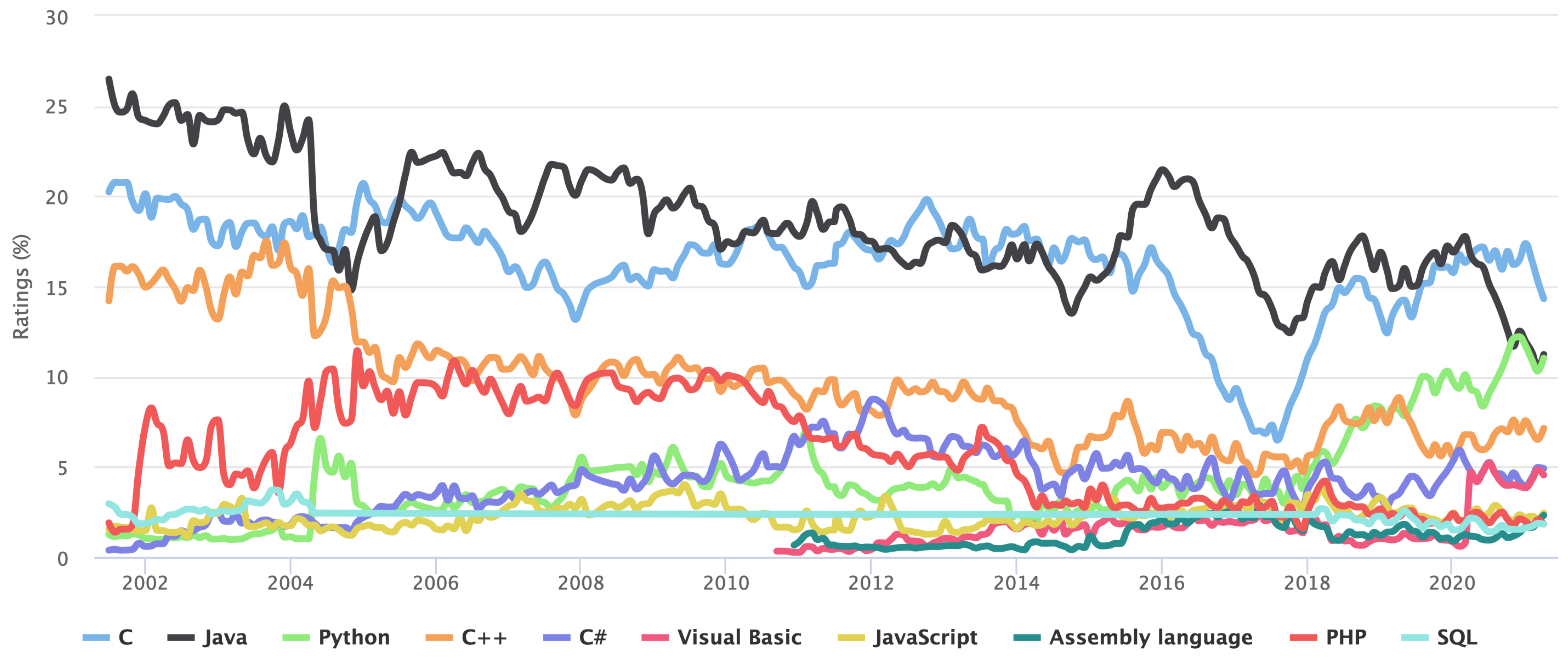
“**The free lunch is over.** We have grown used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year’s model. **If we want our programs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

Language popularity graph (TIOBE)



TIOBE Programming Community Index

Source: www.tiobe.com

ILLINOIS TECH | College of Computing

And the winners are …

- The most popular programming languages are **imperative**, **procedural**, and **object-oriented**
  - **Imperative** programs consist of sequences of statements that read and alter process state
  - **Procedural** languages help us modularize programs by separating logic into different procedures
  - **OOP** lets us bundle together data and procedures that act on them
    - “Big mutable balls”

De facto concurrency model

- Based on shared, freely mutable memory
  - Accessed by explicitly created & managed threads
- Relies on 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*

ILLINOIS INSTITUTE OF TECHNOLOGY
College of Science

Managing complexity

- Implementing correct concurrent behavior via locks is hard!
  - Race conditions are among the most insidious types of bugs
    - Hard to detect, replicate, and fix — lots of extra program complexity (on top of existing application logic)
- **Testing** is a natural way to deal with complexity
  - Idea: write comprehensive test suites to help detect and eliminate concurrency-related bugs, and ensure they stay gone

"… 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**."

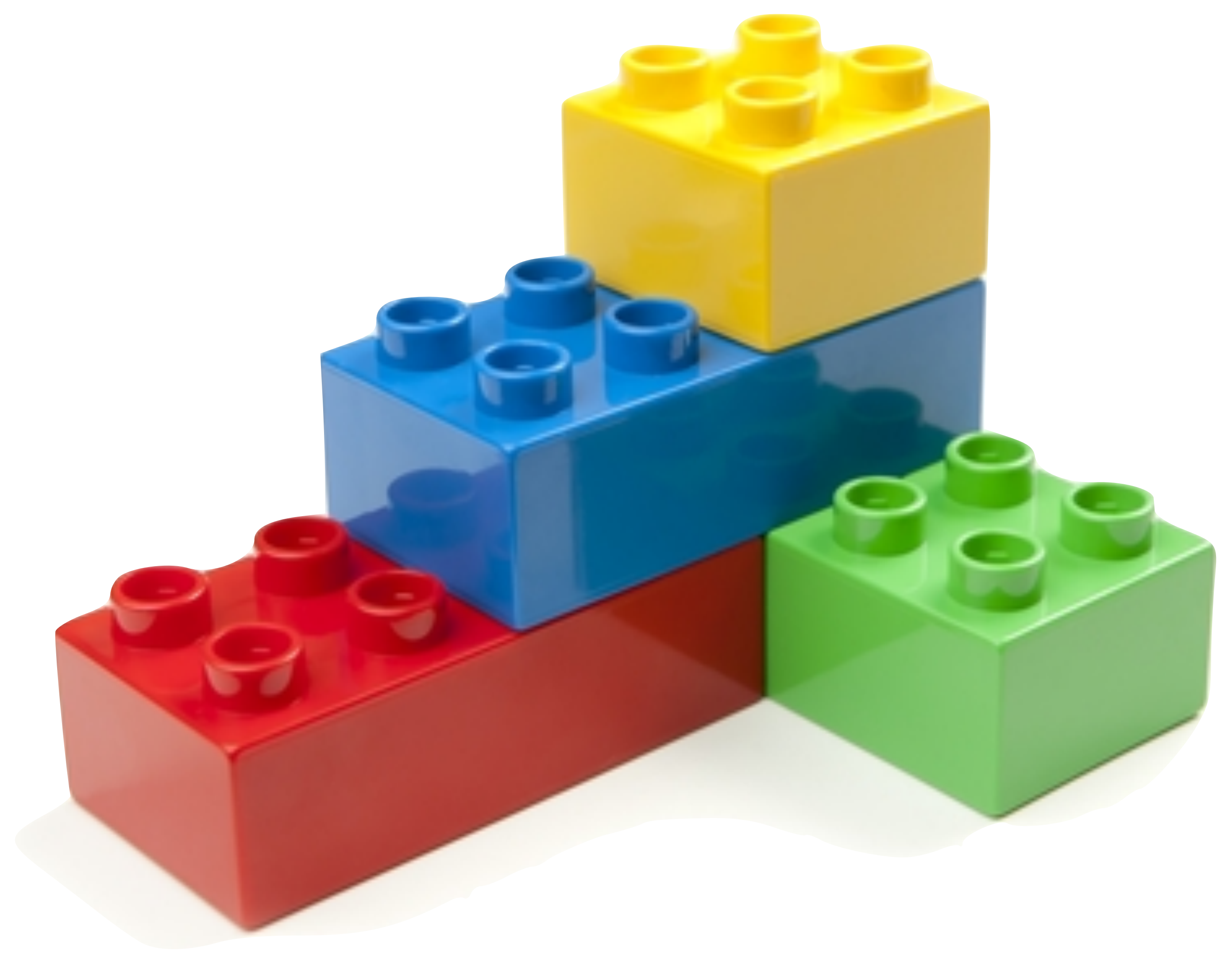
- Ben Moseley and Peter Marks, *Out of the Tar Pit*

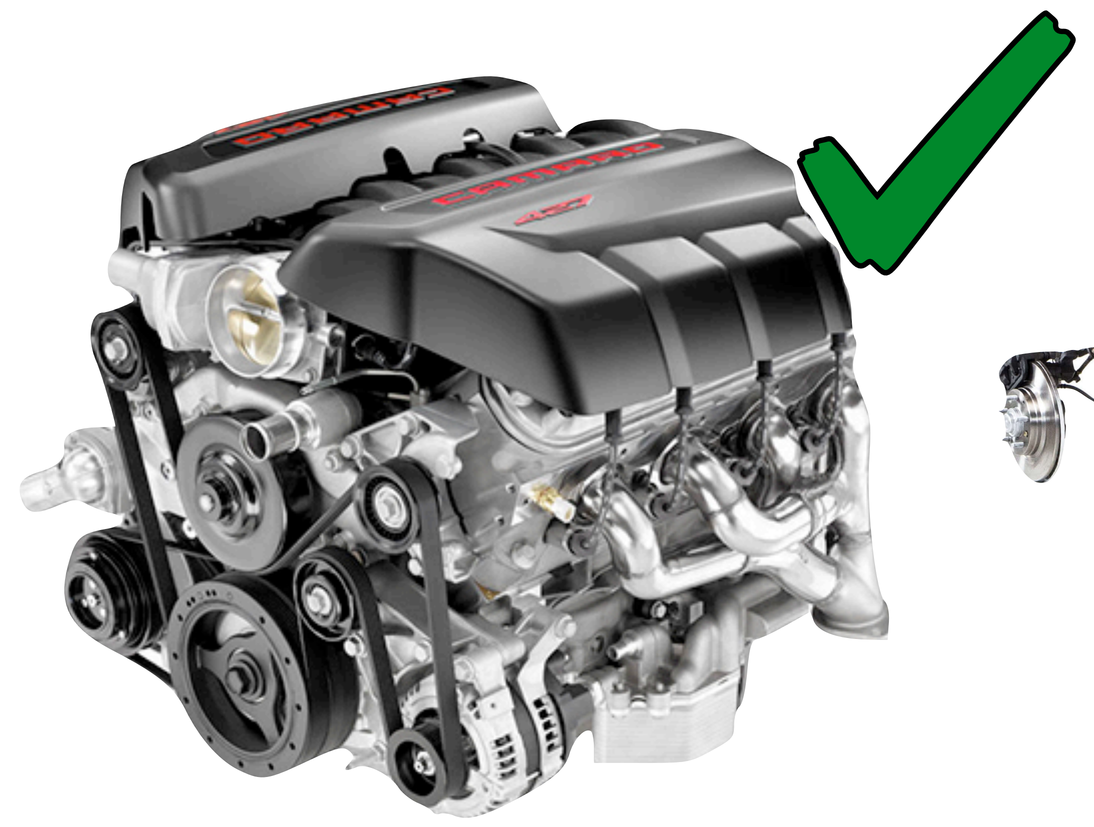
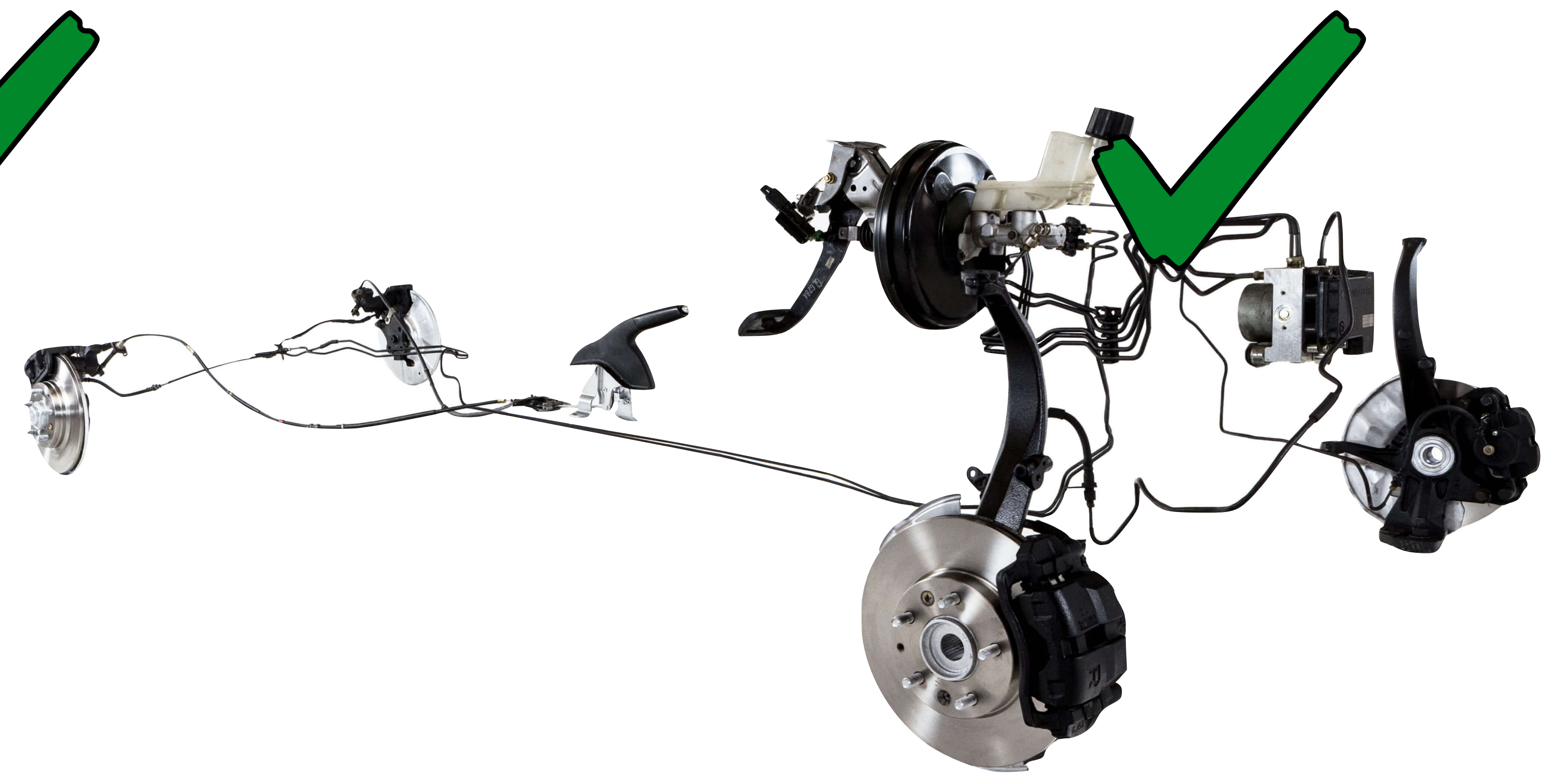
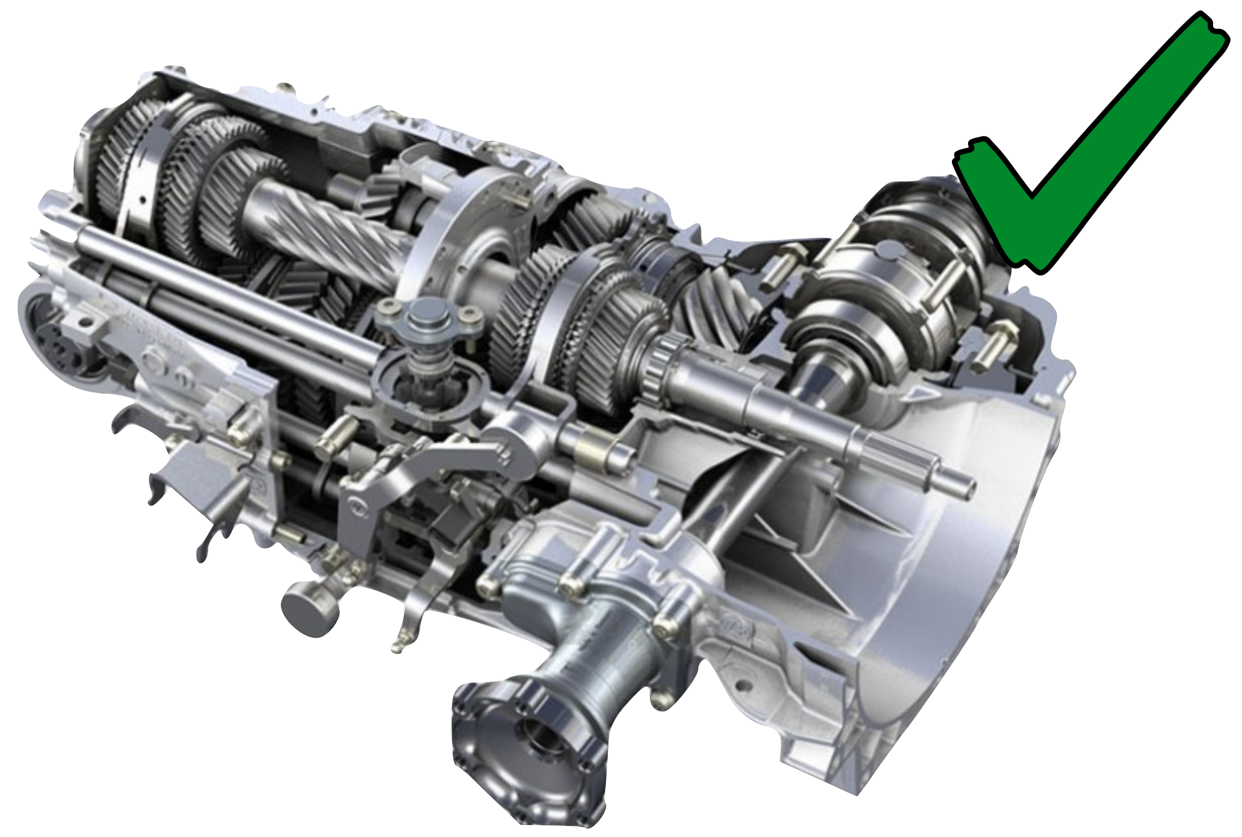
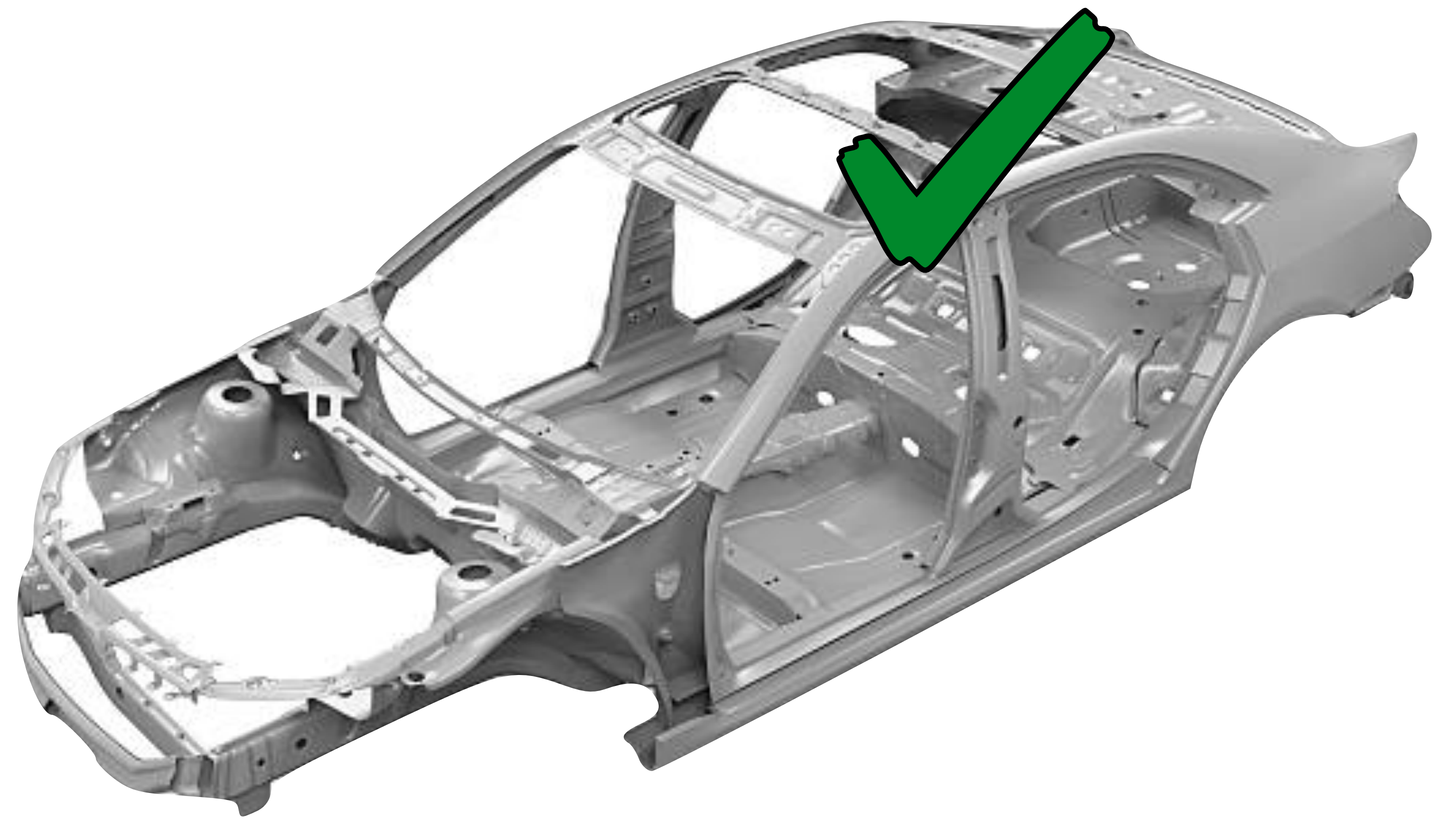
“**Concurrency** also affects testing … **Running a test in the presence of concurrency with a known initial state and set of inputs tells you nothing at all about what will happen the next time you run that very same test with the very same inputs and the very same starting state**. . . and things can’t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

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

Composability

- A key idea in computer science and software engineering is **reusability**
  - An algorithm or program, once developed, can be reused for many different problems/inputs
- Just as important is **composability**
  - We can take independently developed and tested algorithms/modules and **compose** them together to build larger ones



=

Live to Rally
217
BUG RAL
CALIFORNIA
BUGS RALLY EXHIBITION

“A fundamental problem with locks is that they are **not composable**. You can’t take two correct lock-based pieces of code, combine them, and know that the result is still correct. Modern software development relies on the ability to compose libraries into larger programs, and so it is a serious difficulty that **we cannot build on lock-based components without examining their implementations**.”

- Herb Sutter and James Larus,
Software and the Concurrency Revolution

"… 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*

We deserve better!

- Lock-based concurrent programming defeats composability
  - Makes building large, complex systems exponentially more difficult
- Root of the problem: low-level thread and lock APIs
  - Forces us to explicitly manage concurrency, often commingling synchronization and application logic
- We want (and deserve) higher-level APIs!

"Most of today's languages offer explicit programming at the level of **threads and locks**. These abstractions are **low-level and difficult to reason about** systematically. Because these constructs are a poor basis for building abstractions, they encourage multithreaded programming with its **problems of arbitrary blocking and reentrancy**.

Higher-level abstractions allow programmers to express tasks with **inherent concurrency**, which a runtime system can then combine and schedule to fit the hardware on the actual machine. This will enable applications that **perform better on newer hardware**."

- Herb Sutter and James Larus,
Software and the Concurrency Revolution

§ Alternative concurrent programming paradigms

- Software transactional memory (STM)

STM: Overview

- Shared memory objects exist, but access to them is vetted and managed by runtime software
- No explicit locking and/or fine-grained synchronization logic
- Shared objects may only be manipulated within *transactions*
  - Akin to database transactions
  - STM automatically ensures key transactional properties

Transaction properties

- ACI~~D~~ properties
  - **Atomicity**: all operations in the transaction succeed or fail as a unit
  - **Consistency**: results of transactions are validated before being committed
  - **Isolation**: intermediate results of a transaction aren’t visible to others

E.g., bank account transactions

- Consider withdrawing X from account A and depositing it into account B
  - **Atomicity**: either withdrawal & deposit both succeed or fail
    - Otherwise, maybe X is withdrawn and not deposited!
  - **Consistency**: e.g., ensure that A is not overdrawn at end of transaction before changes are committed
  - **Isolation**: intermediate state (X is withdrawn and not deposited, or deposited and not withdraw) is not visible
    - Otherwise, balance inquiry taken mid-transaction could be misleading!

E.g., bank transactions (pthreads)

```
typedef struct {
    long balance;
} account_t;

account_t accounts[N_ACCS];

int withdraw(account_t *acc, long amount) {
    if (amount > acc->balance)
        return 0;
    acc->balance = acc->balance - amount;
    return 1;
}

void deposit(account_t *acc, long amount) {
    acc->balance = acc->balance + amount;
}
```

```
void *do_transactions(void *arg) {
    for (int i=0; i<N_TRANS; i++) {
        int from = random() % N_ACCS,
            to   = random() % N_ACCS;
        long amt = random() % 100;
        if (withdraw(&accounts[from], amt)) {
            deposit(&accounts[to], amt);
        }
    }
}

main() {
    for (int i=0; i<N_THREADS; i++)
        pthread_create(&tids[i], NULL,
                       do_transactions, NULL);
}
```

E.g., bank transactions (pthreads)

```
void *do_transactions(void *arg) {
    for (int i=0; i<N_TRANS; i++) {
        int from = random() % N_ACCS,
            to   = random() % N_ACCS;
        long amt = random() % 100;
        if (withdraw(&accounts[from], amt)) {
            deposit(&accounts[to], amt);
        }
    }
}

main() {
    for (int i=0; i<N_THREADS; i++)
        pthread_create(&tids[i], NULL,
                       do_transactions, NULL);
}
```

Two runs, N_THREADS=10, N_ACCS=10, N_TRANS=100000:

```
Start:
  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
  Total: 10000

End:
  1927 -75 806 1045 -39 1671 960 580 478 571
  Total: 7924
```

```
Start:
  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
  Total: 10000

End:
  856 128 650 3992 107 1016 182 -25 2945 774
  Total: 10625
```

E.g., bank transactions (pthreads)

```c
typedef struct {
    long balance;
    pthread_mutex_t lock;
} account_t;

void *do_transactions(void *arg) {
  for (int i=0; i<NUM_TRANS; i++) {
    int from = random() % NUM_ACCS,
        to   = random() % NUM_ACCS;
    long amt = random() % 100;
    pthread_mutex_lock(&accounts[from].lock);
    pthread_mutex_lock(&accounts[to].lock);
    if (withdraw(&accounts[from], amt)) {
      deposit(&accounts[to], amt);
    }
    pthread_mutex_unlock(&accounts[from].lock);
    pthread_mutex_unlock(&accounts[to].lock);
  }
}
```

N_THREADS=3, N_ACCS=10, N_TRANS=1000:

```
Start:
  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
  Total: 10000

End:
  910 482 926 682 294 2090 346 24 2478 1768
  Total: 10000
```

N_THREADS=10, N_ACCS=10, N_TRANS=1000:

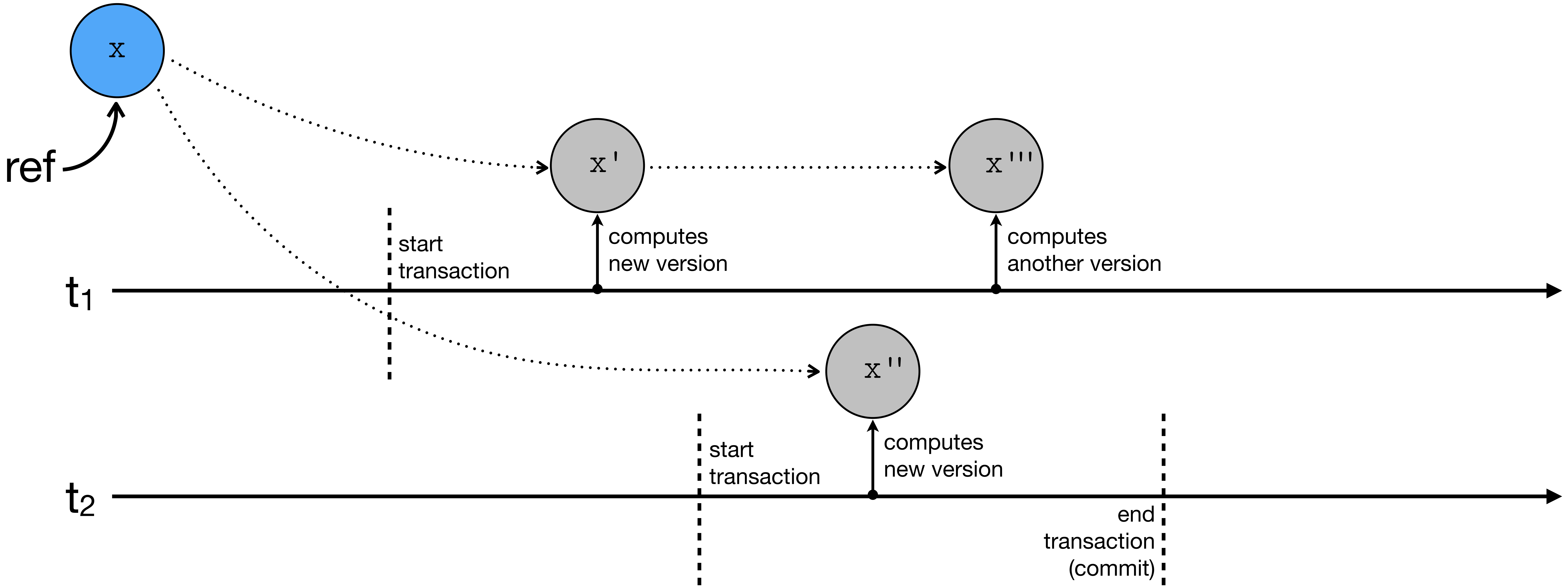
```
Start:
  1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
  Total: 10000

                                          ... (hangs)
```

deadlock!

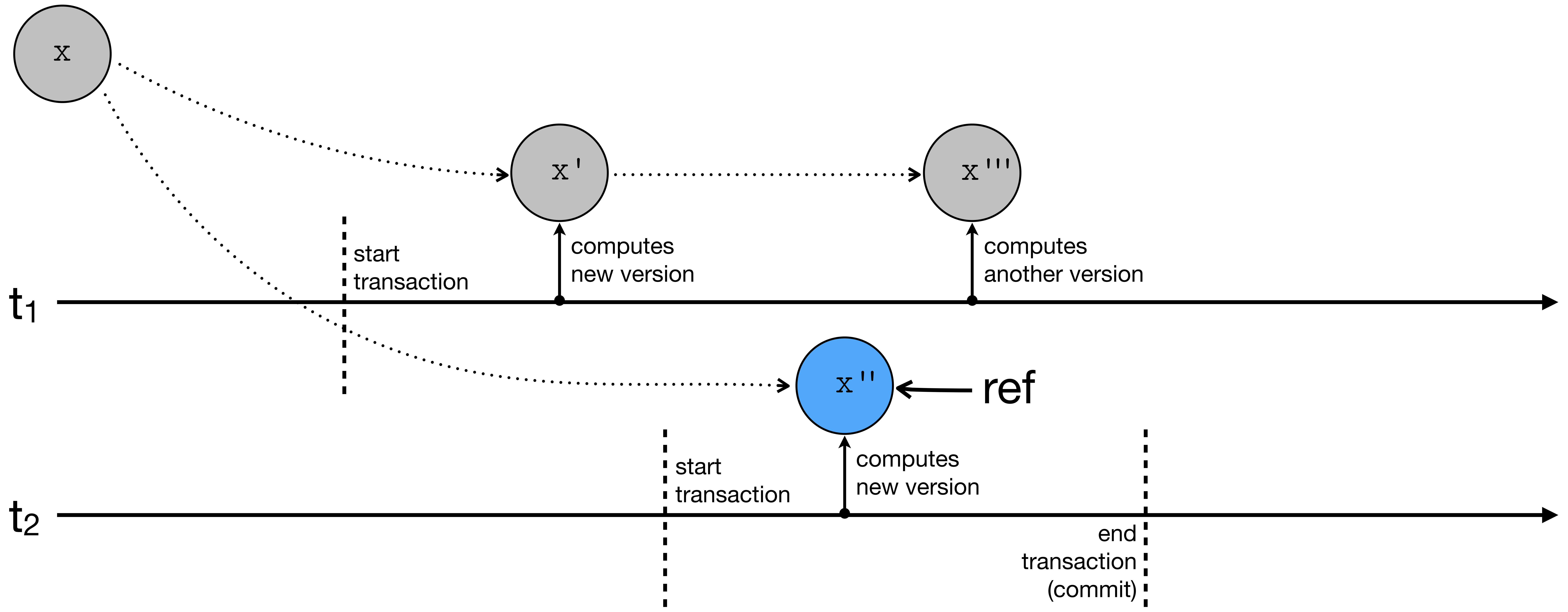
STM in Clojure

- Clojure is a Lisp (what's a Lisp? 🤯) built for the JVM
- Transactional references (`refs`) can only be mutated from within a transaction (bounded by `dosync`)
- Clojure STM is implemented using Multiversion Concurrency Control (MCC)
  - Transactions run concurrently, and each optimistically computes new versions of values based on original references
    - Maximizes concurrency, but may require transaction restarts!

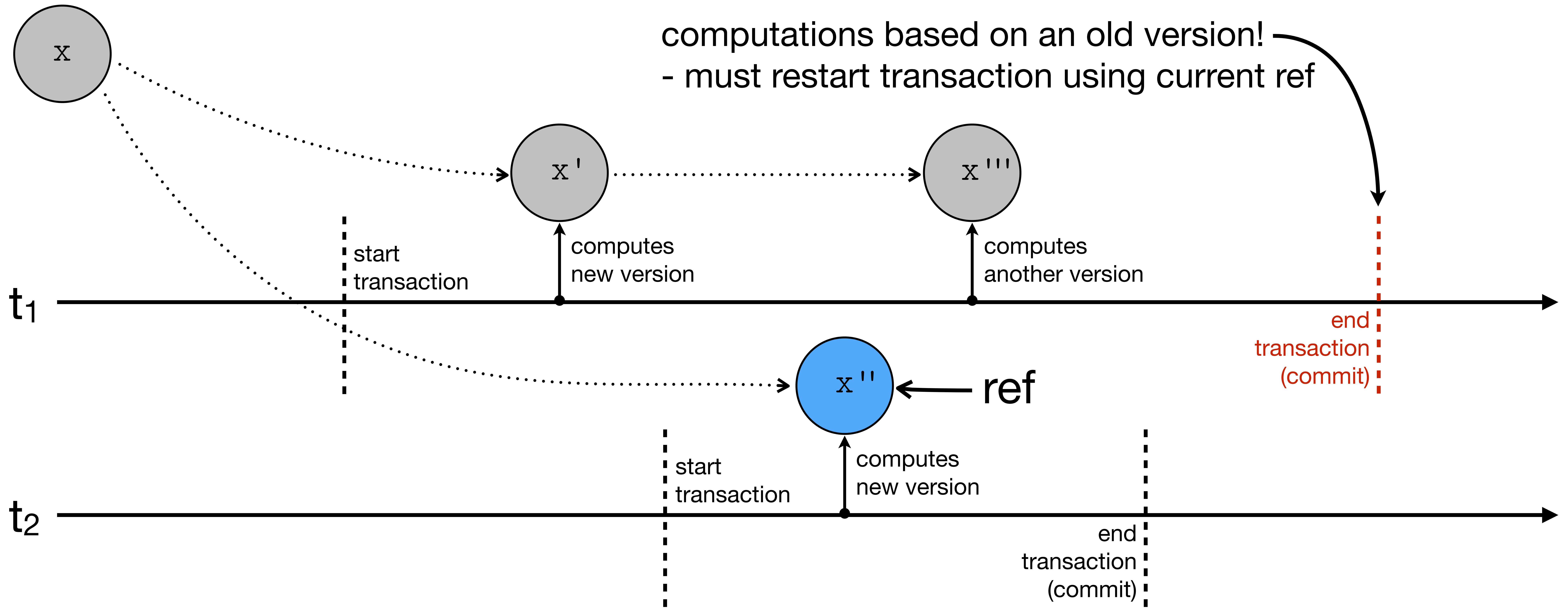
STM in Clojure



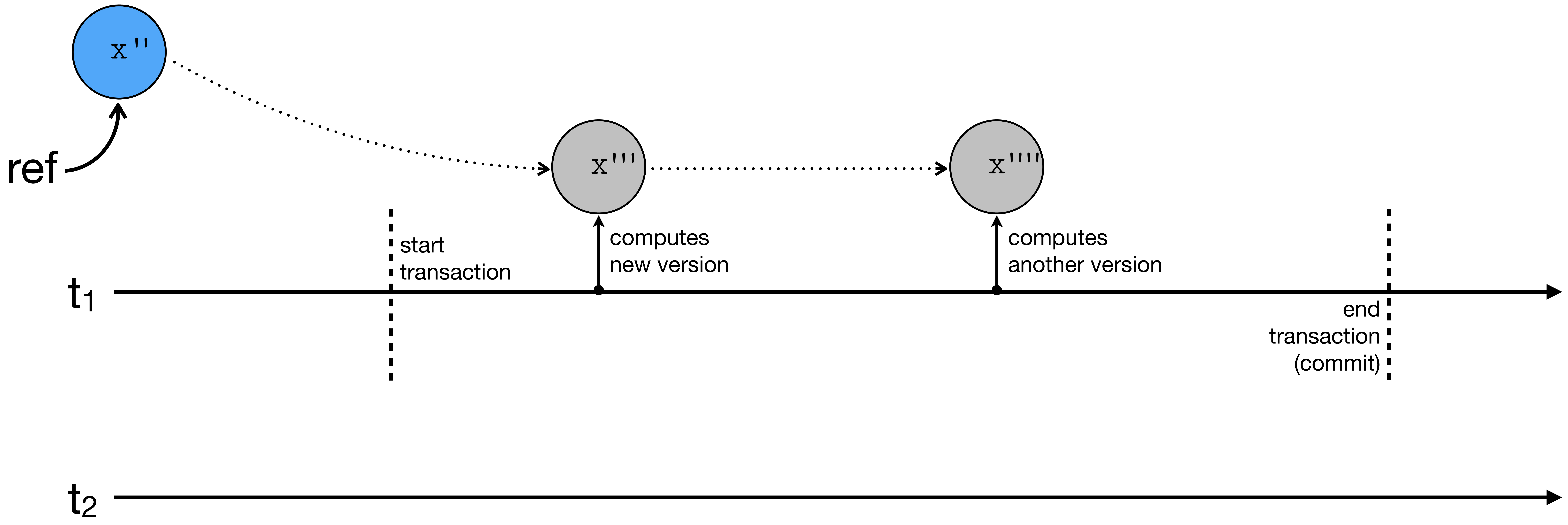
STM in Clojure



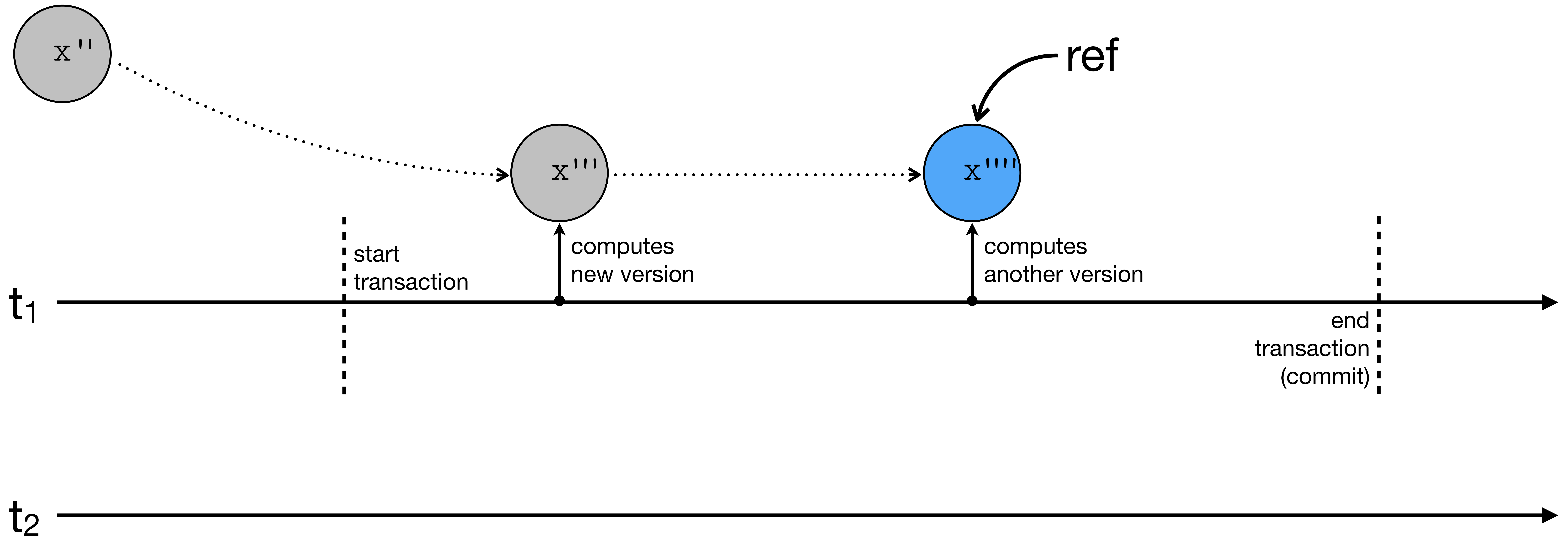
STM in Clojure



STM in Clojure



STM in Clojure



E.g., bank transactions (STM)

```
(defstruct account :balance)

(def accounts
  (repeatedly naccounts
    #(ref (struct account 1000)
          :validator (fn [acc]
                       (>= (:balance acc) 0)))))

(defn withdraw [acc amt]
  (let [bal (:balance acc)]
    (assoc acc :balance (- bal amt))))

(defn deposit [acc amt]
  (let [bal (:balance acc)]
    (assoc acc :balance (+ bal amt))))
```

```
(defn do-transfers [n]
  (dotimes [_ n]
    (let [from (nth accounts (rand-int naccounts))
          to   (nth accounts (rand-int naccounts))
          amt  (rand-int 100)]
      (try
        (dosync ; start transaction
          (alter from withdraw amt)
          (alter to   deposit  amt))
        (catch Exception e)))))

(defn run-transfers [nthreads ntrans]
  (dorun (apply pcalls
                (repeat nthreads
                  #(do-transfers ntrans)))))
```

E.g., bank transactions (STM)

```clojure
(defn do-transfers [n]
  (dotimes [_ n]
    (let [from (nth accounts (rand-int naccounts))
          to   (nth accounts (rand-int naccounts))
          amt  (rand-int 100)]
      (try
        (dosync ; start transaction
          (alter from withdraw amt)
          (alter to   deposit  amt))
        (catch Exception e)))))

(defn run-transfers [nthreads ntrans]
  (dorun (apply pcalls
                (repeat nthreads
                  #(do-transfers ntrans)))))
```

Two runs with nthreads=10, naccounts=10, ntrans=100000:

```
Start:
  (1000 1000 1000 1000 1000 1000 1000 1000 1000 1000)
  10000

End:
  (2827 1443 931 1296 93 1797 122 5 396 1090)
  10000
```

```
Start:
  (1000 1000 1000 1000 1000 1000 1000 1000 1000 1000)
  10000

End:
  (14 1848 1290 391 269 1654 641 2307 330 1256)
  10000
```

STM Pros/Cons

- Language/Runtime takes care of synchronization issues
  - Removes burden from programmer
- But this “magic” has a cost!
  - If optimistic (as with MCC), transaction restarts waste time/energy
  - If pessimistic (e.g., with locking), may not maximize concurrency

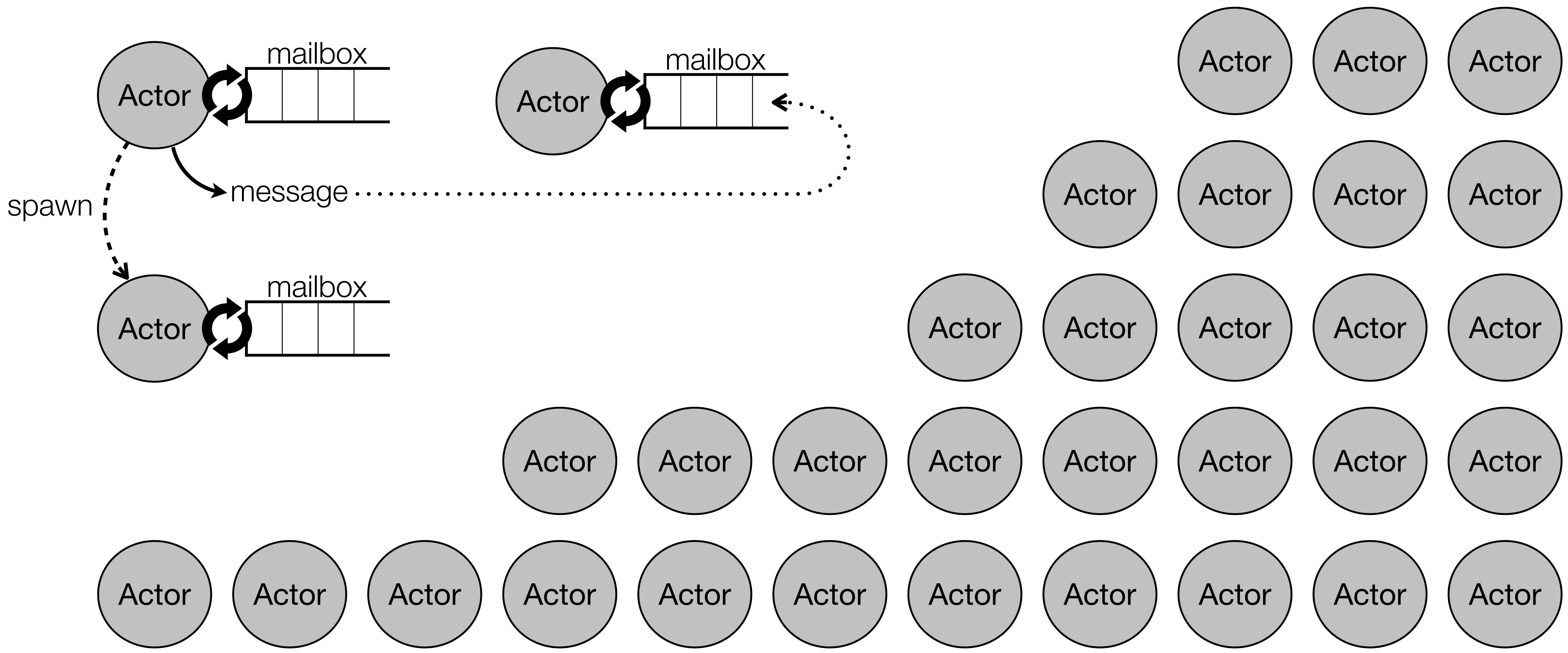
§ Message-passing

ILLINOIS TECH | College of Computing

Message-passing: Overview

- No shared memory between threads (aka “actors”)
  - Message-passing paradigm also known as the “actor model”
- Actors communicate by sending each other messages
  - Messages are queued in per-actor mailboxes
  - Actors can create other actors, compute results, send messages
  - Actors run concurrently (possibly in parallel) and share nothing

Message-passing



Erlang

- Language/Platform 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
- Spawning actors (“processes”) is cheap (scales to millions of processes)
  - Processes communicate via message passing (FIFO mailboxes)
  - Essential model: client/server

Erlang messaging syntax

- Creating processes:

```
Pid = spawn(Fun, Args)
```

- Sending messages (asynchronous):

```
Pid ! Message
```

- Receiving messages (synchronous):

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

Erlang “servers”

- Basic server template:

```
   loop() ->
       receive                                % block for a message in the mailbox
pattern match  terminate -> done;             % “terminate” and “done” are atoms (literals)
(first is taken) Message -> process(Message), % capitalized symbols like “Message” are variables
                            loop()            % recurse to “loop” server body
       end.
```

- Server with “state”:

```
loop(State) ->
    receive
        terminate -> done;
        Message   -> loop(process(Message, State))  % “process” returns new value of State
    end.
```

E.g., bank transactions (Erlang)

```erlang
account(Balance) ->
  receive
    {balance, Pid} ->
      Pid ! Balance,
      account(Balance);
    {deposit, Amount} ->
      account(Balance + Amount);
    {withdraw, Amount} ->
      if
        Balance >= Amount -> account(Balance - Amount);
        true -> account(Balance)
      end;
    {transfer, ToPid, Amount} ->
      if
        Balance >= Amount -> ToPid ! {deposit, Amount},
                             account(Balance - Amount);
        true -> account(Balance)
      end
  end.
```

```erlang
start() ->
  Acc1Pid = spawn(account, [1000]),
  Acc2Pid = spawn(account, [1000]),
  Acc1Pid ! {transfer, Acc2Pid, 250},
  sleep(100), % delay for completion
  Acc1Pid ! {balance, self()},
  Acc2Pid ! {balance, self()}.
```

```
> start().

> flush().  % flush shell's mailbox
Shell got 750
Shell got 1250
```

Projects in Erlang

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

Message-passing Pros/Cons

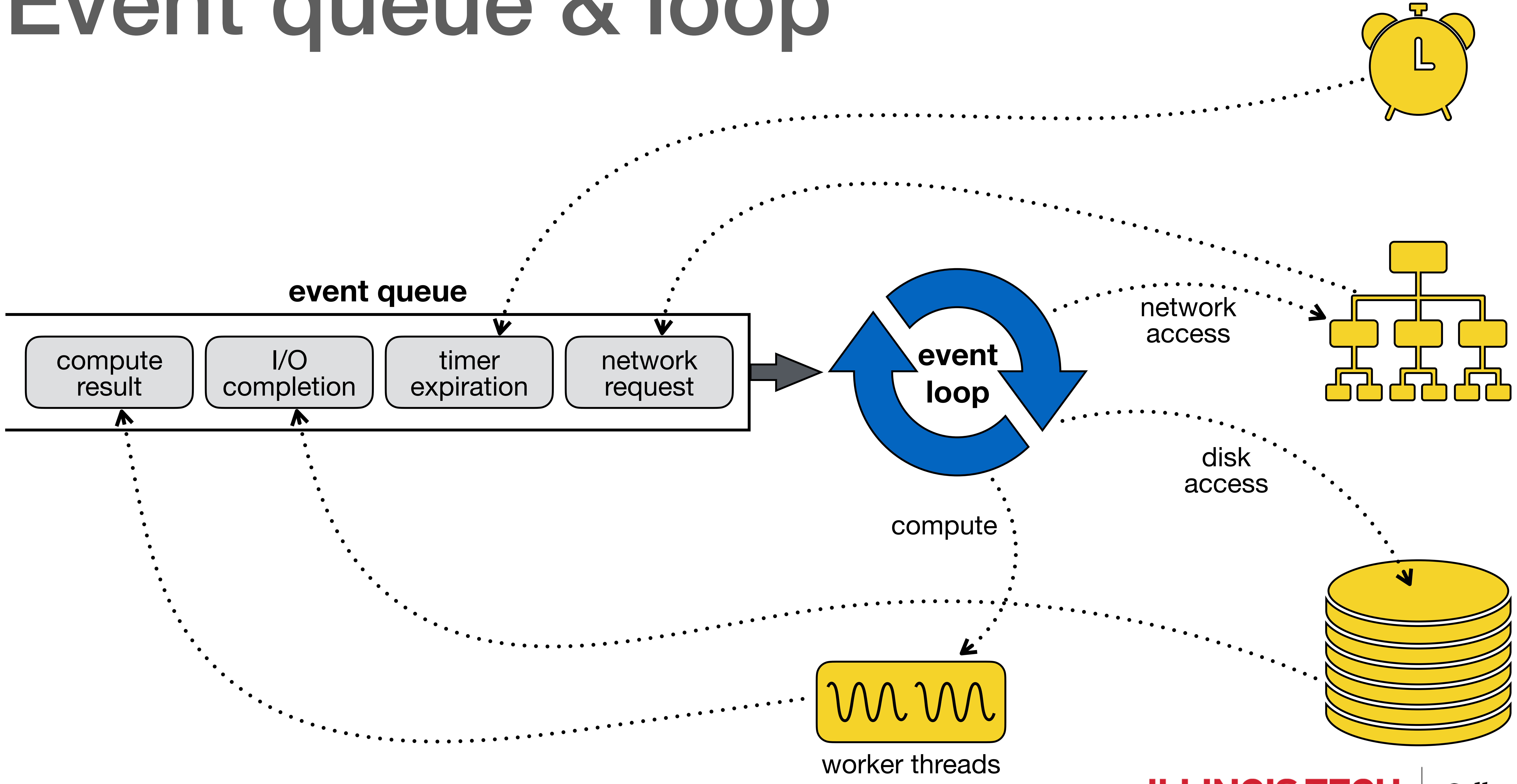
- Scales well, helping to maximize opportunities for concurrency
- Lack of shared state makes (local) reasoning easy
- But synchronization issues do not magically go away!
  - Careful coordination of actors is necessary (deadlock is still possible)
  - Typically addressed by using “standard” communication protocols
- Mailboxes may grow large, and message copying is potentially inefficient

§ Event-driven paradigm

Event-driven: Overview

- Application logic is driven by **events** and associated **event handlers**
- Events are asynchronously generated by a variety of event sources, and entered into an **event queue**
  - An **event loop** draws events from the queue and runs their handlers
    - The event loop and all its handlers are run in a single thread, which simplifies logic and eliminates concurrency-related issues

Event queue & loop



No blocking the loop!

- It is critical that handlers executed from the event loop do not block
  - Otherwise the handling of all pending events will be delayed
- All I/O operations should make use of non-blocking APIs, or be delegated to separate worker threads
  - Results are delivered to the event queue (or checked directly by the event loop) and processed in a single-threaded manner
- Compute-intensive operations should also be delegated to worker threads

Event-specific handlers

- How to specify handlers to execute for all possible event types?
- E.g., consider an incoming network request for records from a database
  - Requires multiple steps / intermediate events:

    1. Connect to database → Event: Connection established
    2. Execute SQL query → Event: SQL result
    3. Process result (if computation required) → Event: Compiled records
    4. Deliver response over network → Event: Delivery success/failure

Central mechanism: Callback

- Instead of registering all possible handlers in advance, specify **callback functions** for each action that generates an event
  - When an action completes, it schedules the callback function to be called (with the resulting event) from the event loop
- Callback functions are often specified as lambdas / anonymous functions
  - Syntactic sugar: Promises and Async/Await

E.g., JavaScript / Node.js

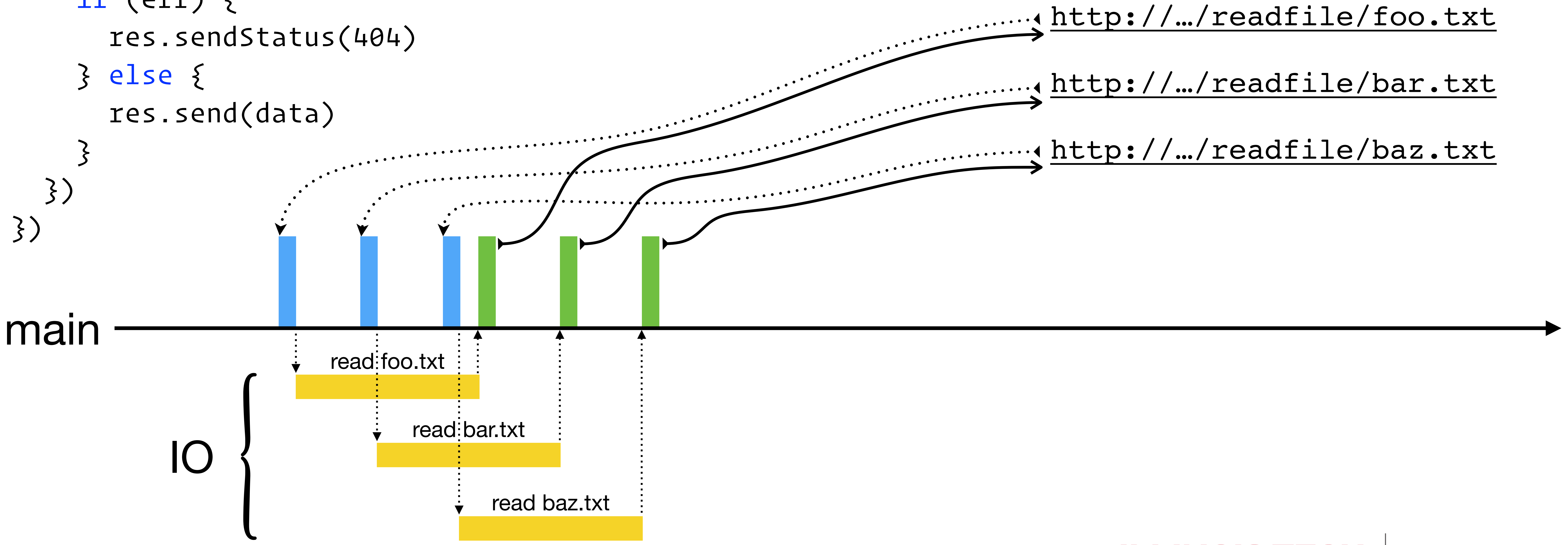
- Focus on web applications (front and back end)
  - Main event queue & thread optimized for responsiveness

```
webapp.get('/readfile/:filename', function (req, res) {
  const filename = req.params.filename
  fs.readFile(filename, 'utf8', function (err, data) {
    if (err) {
      res.sendStatus(404)
    } else {
      res.send(data)
    }
  })
})
```

callback functions

E.g., JavaScript / Node.js

```
webapp.get('/readfile/:filename', function (req, res) {
  const filename = req.params.filename
  fs.readFile(filename, 'utf8', function (err, data) {
    if (err) {
      res.sendStatus(404)
    } else {
      res.send(data)
    }
  })
})
```



http://…/readfile/foo.txt
http://…/readfile/bar.txt
http://…/readfile/baz.txt
main
read foo.txt
read bar.txt
read baz.txt
IO

ILLINOIS TECH | College of Computing

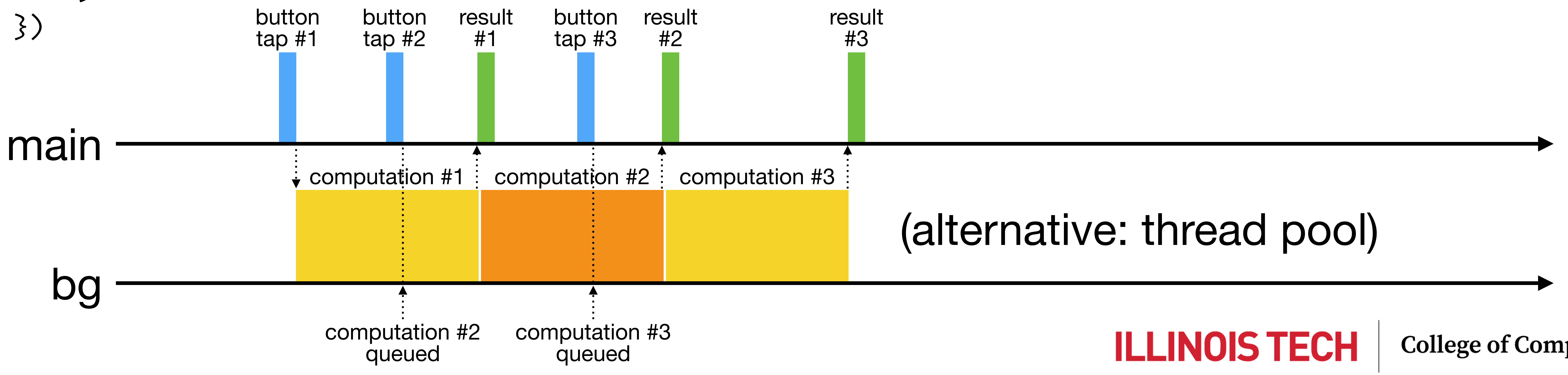
E.g., Swift

- Focus on native iOS/Mac app development
  - All UI updates/events processed in the main event queue & thread
    - Important to not to block the main thread to keep UI responsive!

```
Button("Start a lengthy operation ...", action: {
    DispatchQueue.global(qos: .userInitiated).async {
        print("This runs in a background thread")
        sleep(1); // simulate long-running computation
        DispatchQueue.main.async {
            // display result
            print("This runs back in the main event-handling thread again")
        }
    }
})
```

E.g., Swift

```
Button("Start a lengthy operation ...", action: {
    DispatchQueue.global(qos: .userInitiated).async {
        print("This runs in a background thread")
        sleep(1); // simulate long-running computation
        DispatchQueue.main.async {
            // display result
            print("This runs back in the main event-handling thread again")
        }
    }
})
```



(alternative: thread pool)

ILLINOIS TECH | College of Computing

Event-driven Pros/Cons

- Single-threaded event loop doesn't take advantage of parallel computation
  - With worker threads and separate queues, we have (mostly) the same issues as message-passing
- Relies on non-blocking system calls
  - Without them, must again resort to multithreaded code
- Asynchronous code is generally harder to write and understand than synchronous/sequential code (search "callback hell" & "pyramid of doom")
  - Modern APIs try to address this with Promises and other constructs

§ Summary

Summary

- Concurrent (and parallel) programming is hard, but arguably necessary
- Classical lock-driven concurrent programs are complex, error-prone, and non-composable
- There are many alternatives to lock-based concurrent programming, including STM, the actor model, and event-driven programming
  - None are perfect!
    - Do your research to understand what best suits your problem domain