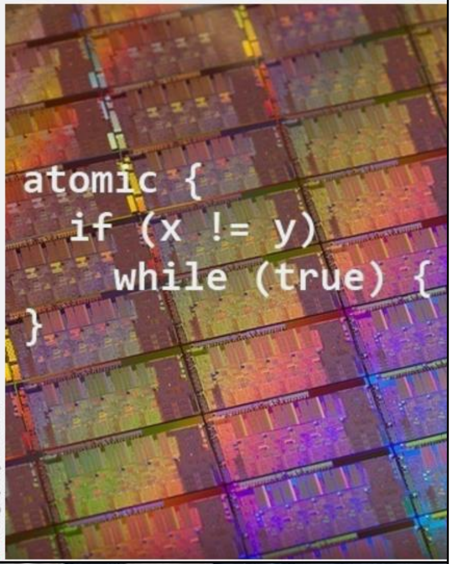
TORSTEN HOEFLER

Parallel Programming
Transactional Memory & Programming based on Message Passing

Transactional memory going mainstream with Intel Haswell

Transactional memory is a promising technique f

PETER BRIGHT - 2/9/2012, 3:10 AM



Errata prompts Intel to disable TSX in Haswell, early Broadwell CPUs

by Scott Wasson — 1:28 PM on August 12, 2014

The TSX instructions built into Intel's Haswell CPU cores haven't become widely used by everyday software just yet, but they promise to make certain types of multithreaded applications run much faster than they can today. Some of the savviest software developers are likely building TSX-enabled software right about now.

Unfortunately, that work may have to come to a halt, thanks to a bug—or "errata," as Intel prefers to call them—in Haswell's TSX implementation that can cause critical software failures.

I believe my friend David Kanter was first to report this problem via a tweet the other day. Intel revealed the news of the erratum to a group of journalists during briefings in Portland last week. I was among those in attendance and was able to talk with Intel architects about the situation.

The TSX problem was apparently discovered by a software developer outside of Intel, and the company then confirmed the erratum through its own testing. Errata of this magnitude aren't often dis

As is customary in such cases, Intel has disabled the CPU microcode update delivered via new revisions o ensure stable operation for Haswell CPUs, but those TSX's features, including hardware lock elision and r

Software developers who wish to continue working systems to newer firmware revisions—and in doing s corruption or crashes.

Intel Launches Kaby-Lake based Xeons: The E3-1200 v6 Family

54 Comments

by Ian Cutress on March 28, 2017 2:00 PM EST

+ Add A Comment

Posted in CPUs Intel Xeon Enterprise enterprise CPUs E3 Optane E3-1200 v6 E3-1200

...

The high-end E3 v6 parts will have a maximum base frequency of 3.9 GHz base and a 4.2 GHz turbo. All the parts listed have a full 8MB of L3 cache, and either be 72W for non-IGP models or 73W for IGP parts. As with other previous Xeons, these come with ECC memory support, vPro and other technologies Intel files under the professional level. In Intel's presentations, Intel SGX (Software Guard Extensions) are included, however TSX (Transactional Extensions) were not listed

The Consensus Hierarchy

1	Read/Write Registers	
2	getAndSet, getAndIncrement, …	FIFO Queue LIFO Stack
. .		
∞	CompareAndSet, …	Multiple Assignment

Consensus - conclusion

- Consensus is the simplest wait-free problem
  - Easy to define and prove (will come later)

- Consensus number
  - How many threads can objects of class C coordinate (wait-free)?

    Wait-free FIFO queues have consensus number 2

    Test-And-Set, getAndSet, getAndIncrement have consensus number 2

    CAS has consensus number ∞

- Consensus itself is a powerful tool to prove impossibility!
  - Saw it with the FIFO queue
  - Here, we discuss only wait-free

Motivation for

Transactional Memory

Transactional Memory in a nutshell

Motivation: programming with locks is too difficult

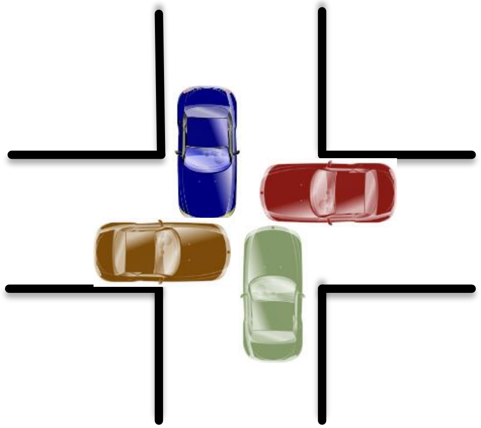
Lock-free programming is even more difficult...

Goal: remove the burden of synchronization from the programmer and place it in the system (hardware / software)

Literature:
-Herlihy Chapter 18.1 – 18.2.
-Herlihy Chapter 18.3. interesting but too detailed for this course.

What is wrong with locking?

Deadlocks: threads attempt to take common locks in different orders

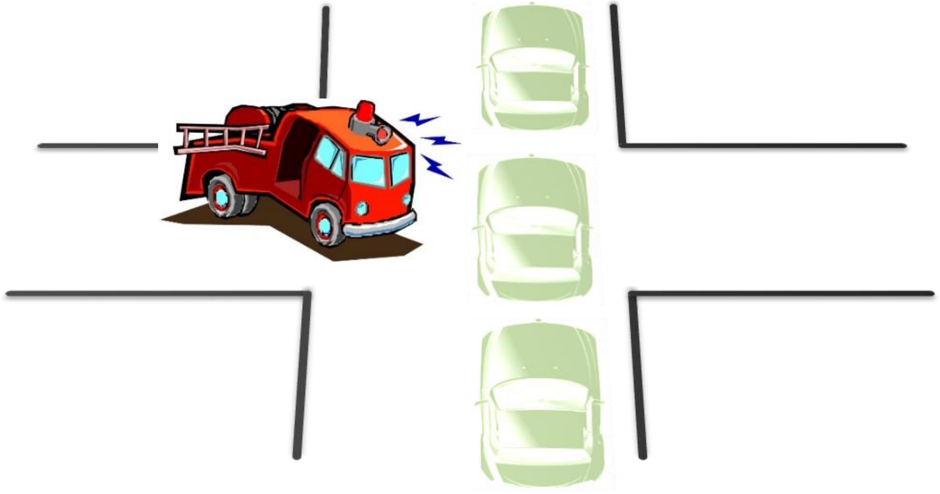


What is wrong with locking?

Convoying: thread holding a resource R is descheduled while other threads queue up waiting for R

What is wrong with locking?

Priority Inversion: lower priority thread holds a resource R that a high priority thread is waiting on



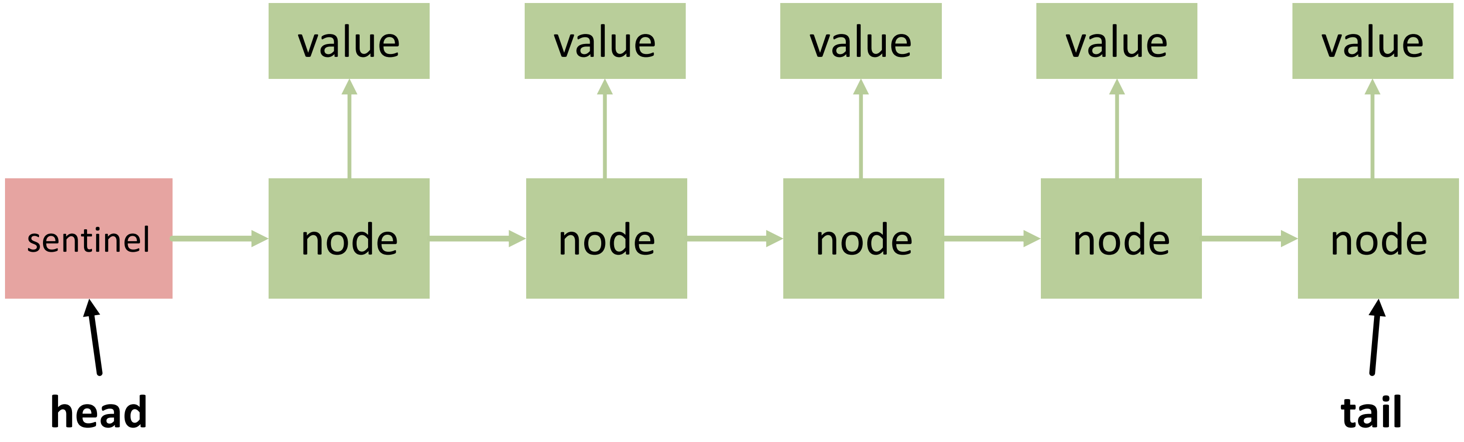
What is wrong with locking?

Association of locks and data established **by convention**.

The best you can do is **reasonably document** your code!

What is wrong with CAS?

Example: Unbounded Queue (FIFO)

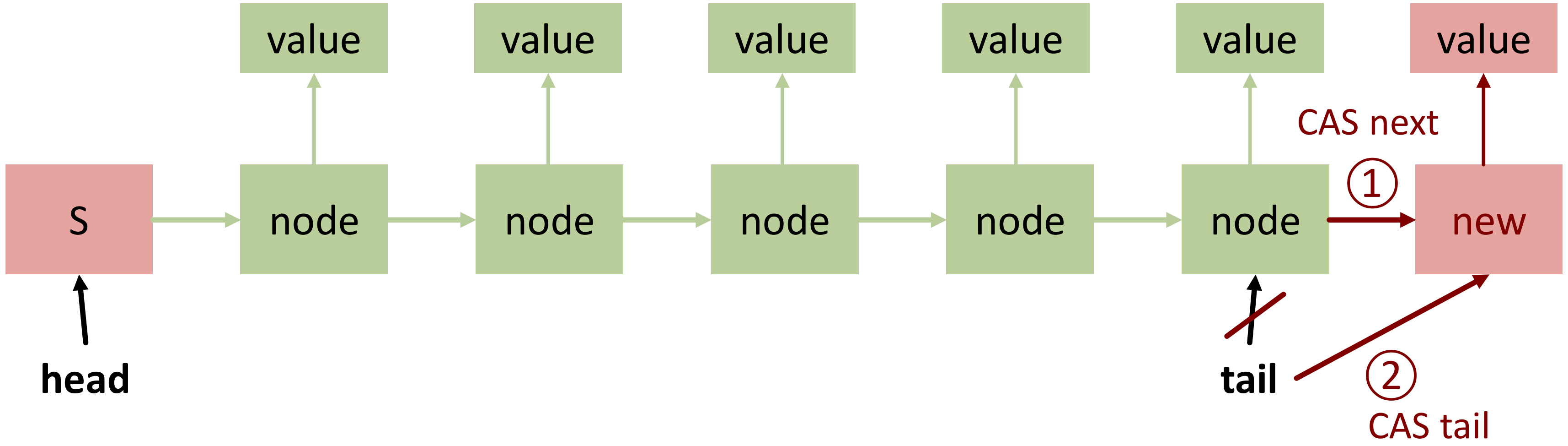



```
public class LockFreeQueue<T> {
   private AtomicReference<Node>
head;
   private AtomicReference<Node>
tail;

   public void enq(T item);
   public T deq();
}
```

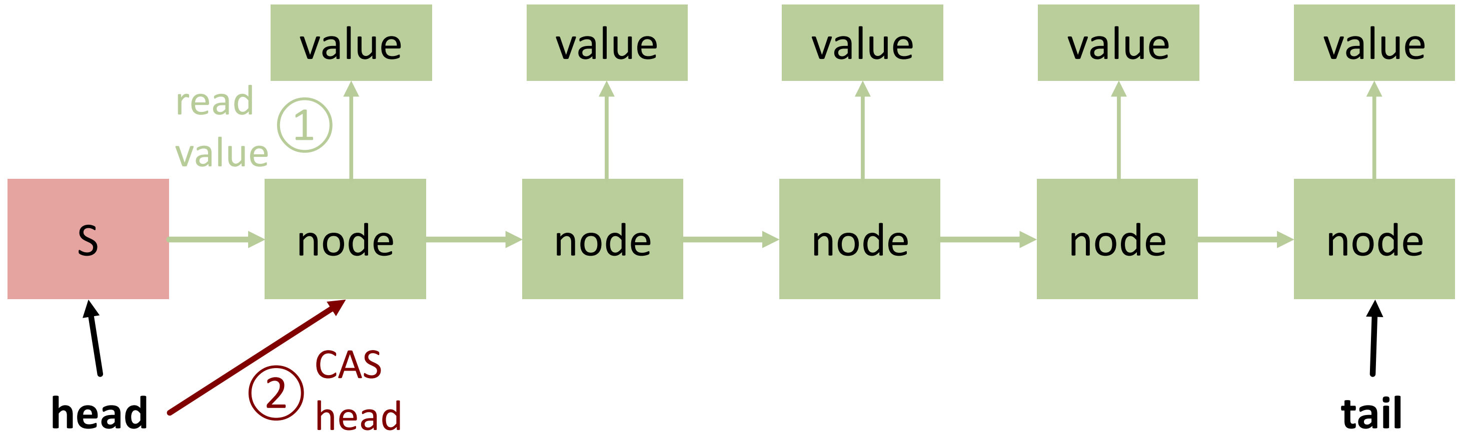
```
public class Node {
  public T value;
  public AtomicReference<Node> next;
  public Node(T v) {
    value = v;
    next = new
AtomicReference<Node>(null);
  }
}
```

Enqueue



Two CAS operations → **half finished enqueue visible** to other processes

Dequeue


value
value
value
value
value
read
value
1
S
node
node
node
node
node
head
2
CAS
head
tail

Code for Enqueue

```
public class LockFreeQueue<T> {
..
   public void enq(T item) {
      Node node = new Node(item);
      while(true){
         Node last = tail.get();
         Node next = last.next.get();
         if (last == tail.get()) {
            if (next == null)
               if (last.next.compareAndSet(next, node)) {
                  tail.compareAndSet(last, node);
                  return;
               }
            else
               tail.compareAndSet(last, next);
         }
      }
   }
}
```

Half finished insert may happen!

Help other processes with finishing operations (→ lock-free)

Code with hypothetical DCAS

```
public class LockFreeQueue<T> {
..
   public void enq(T item) {
      Node node = new Node(item);
      while(true) {
         Node last = tail.get();
         Node next = last.next.get();
         if (multiCompareAndSet({last.next, tail},{next, last},{node, node})
            return;
      }
   }
}
```

This code ensures consistency of both next and last: operation **either fails completely without effect or the effect happens atomically**

More problems: Bank account

```
class Account {
  private final Integer id;       // account id
  private       Integer balance;  // account balance

  Account(int id, int balance) {
      this.id      = new Integer(id);
      this.balance = new Integer(balance);
  }

  synchronized void withdraw(int amount) {
      // assume that there are always sufficient funds...
      this.balance = this.balance – amount;
  }

  synchronized void deposit(int amount) {
      this.balance = this.balance + amount;
  }
}
```

Bank account transfer (unsafe)

```
void transfer_unsafe(Account a, Account b, int amount) {

      a.withdraw(amount);
      b.deposit(amount);
}
```

Transfer does not happen atomically

A thread might observe the withdraw, but not the deposit

Bank account transfer (can cause a deadlock)

```
void transfer_deadlock(Account a, Account b, int amount) {
      synchronized (a) {
            synchronized (b) {
                  a.withdraw(amount);
                  b.deposit(amount);
            }
      }
}
```

Concurrently executing:

- `transfer_deadlock(a, b)`
- `transfer_deadlock(b, a)`

Might lead to a deadlock

Bank account transfer (lock ordering to avoid deadlock)

```
void transfer(Account a, Account b, int amount) {
  if (a.id < b.id) {
      synchronized (a) {
          synchronized (b) {
              a.withdraw(amount);
              b.deposit(amount);
          }
      }
  } else {
      synchronized (b) {
          synchronized (a) {
              a.withdraw(amount);
              b.deposit(amount);
          }
      }
  }
}
```

Bank account transfer (slightly better ordering version)

```
void transfer_elegant(Account a, Account b, int amount) {
      Account first, second;
      if (a.id < b.id) {
          first = a;
          second = b;
      } else {
          first = b;
          second = a;
      }

      synchronized (first) {
          synchronized (second) {
              a.withdraw(amount);
              b.deposit(amount);
          }
      }
}
```

Code for synchronization

Code for the actual operation

Lack of composability

Ensuring ordering (and correctness) is **really hard** (even for advanced programmers)

- rules are ad-hoc, and not part of the program
- (documented in comments at best-case scenario)

Locks are **not composable**

- how can you combine **n** thread-safe operations?
- internal details about locking are required
- big problem, especially for programming “in the large”

Problems using locks (cont'd)

Locks are pessimistic

- worst is assumed
- performance overhead paid every time

Locking mechanism is hard-wired to the program

- synchronization / rest of the program cannot be separated
- changing synchronization scheme → changing all of the program

Solution: atomic blocks (or transactions)

What the programmer actually meant to say is:

```
atomic {
    a.withdraw(amount);
    b.deposit(amount);
}
```

I want these operations
to be performed atomically!

→ This is the idea behind transactional memory
also behind locks, isn't it? The difference is the *execution*!

Transactional Memory (TM)

Programmer explicitly defines atomic code sections

Programmer is concerned with:

what: what operations should be atomic

but, **not how**: e.g., via locking
the how is left to the system (software, hardware or both)

(declarative approach)

TM benefits

- simpler and less error-prone code
- higher-level (declarative) semantics (what vs. how)
- composable
- analogy to garbage collection
  (Dan Grossman. 2007. *"The transactional memory / garbage collection analogy"*. SIGPLAN Not. 42, 10 (October 2007), 695-706.)
- optimistic by design
  (does not require mutual exclusion)

TM semantics: Atomicity

changes made by a transaction are

made visible atomically

other threads preserve either the initial or the final state, but not any intermediate states

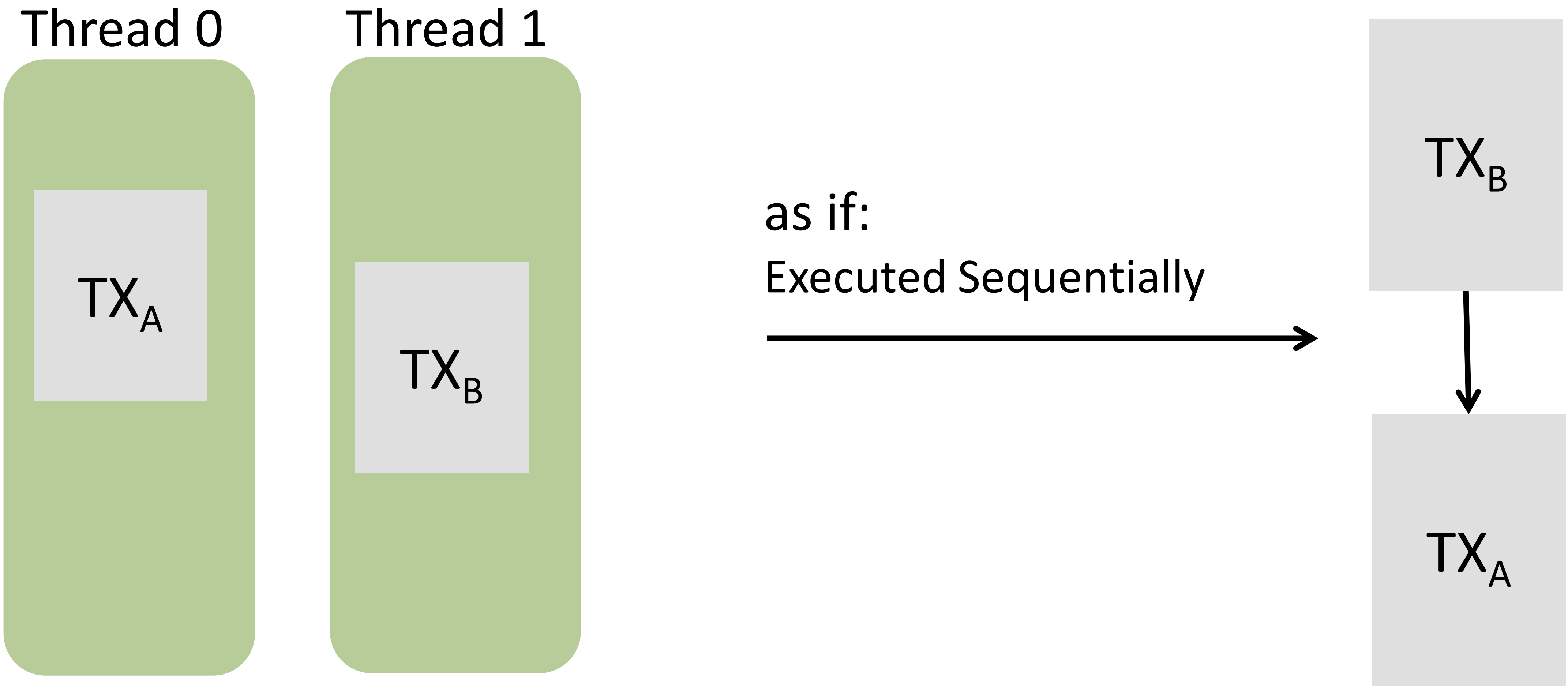
Note: locks enforce atomicity via mutual exclusion, while transactions do not require mutual exclusion

TM semantics: Isolation

Transactions run in isolation

- while a transaction is running, effects from other transactions are not observed
- as if the transaction takes a snapshot of the global state when it begins and then operates on that snapshot

Serializability



(transactions appear serialized)

Transactions in databases

Transactional Memory is heavily inspired by database transactions

ACID properties in database transactions:

- Atomicity
- Consistency (database remains in a consistent state)
- Isolation (no mutual corruption of data)
- Durability (e.g., transaction effects will survive power loss → stored in disk)

How to implement TM?

Which are missing?

Big lock around all atomic sections

- gives (nearly all) desired properties, but not scalable
- not done in practice for obvious reasons

Keep track of operations performed by each transaction

- concurrency control
- system ensures atomicity and isolation properties

What happens when a conflict occurs?

Conflict example: a transaction (not yet committed) has read a value that was changed by a transaction that has committed

Initially: a = 0

TX_A

```
atomic {
    …
    x = a // read a
    if (x == 0) {
        …
    } else {
        …
    }
}
```

TX_B

```
atomic {
    …
    a = 10 // write a
    …
}
```

What happens when a conflict occurs?

Conflict example: a transaction (not yet committed) has read a value that was changed by a transaction that has committed

Initially: a = 0

TX_A

```
atomic {
    …
    x = a // read a
    if (x == 0) {
        …
    } else {
        …
    }
}
```

started

TX_B

```
atomic {
    …
    a = 10 // write a
    …
}
```

not started

What happens when a conflict occurs?

Conflict example: a transaction (not yet committed) has read a value that was changed by a transaction that has committed

Initially: a = 0

TX$_A$
```
atomic {
    …
    x = a // read a
    if (x == 0) {
        …
    } else {
        …
    }
}
```

read value 0

TX$_B$
```
atomic {
    …
    a = 10 // write a
    …
}
```

not started

What happens when a conflict occurs?

Conflict example: a transaction (not yet committed) has read a value that was changed by a transaction that has committed

Initially: a = 0

TX_A

```
atomic {
    …
    x = a // read a
    if (x == 0) {
        …
    } else {
        …
    }
}
```

read value 0

TX_B

```
atomic {
    …
    a = 10 // write a
    …
}
```

started

What happens when a conflict occurs?

Conflict example: a transaction (not yet committed) has read a value that was changed by a transaction that has committed

Initially: a = 0

TX_A

```
atomic {
    …
    x = a // read a
    if (x == 0) {
        …
    } else {
        …
    }
}
```

read value 0

TX_B

```
atomic {
    …
    a = 10 // write a
    …
}
```

update a locally not visible to other transactions

What happens when a conflict occurs?

Conflict example: a transaction (not yet committed) has read a value that was changed by a transaction that has committed

Initially: a = 0

TX_A

```
atomic {
    …
    x = a // read a
    if (x == 0) {
        …
    } else {
        …
    }
}
```

read value 0

TX_B

```
atomic {
    …
    a = 10 // write a
    …
}
```

commited changes visible to other transactions

What happens when a conflict occurs?

Conflict example: a transaction (not yet committed) has read a value that was changed by a transaction that has committed

Initially: a = 0

TX_A

```
atomic {
    …
    x = a // read a
    if (x == 0) {
        …
    } else {
        …
    }
}
```

commited
can this transaction now be placed after TX_B in the serialization order?

TX_B

```
atomic {
    …
    a = 10 // write a
    …
}
```

commited
changes visible to other transactions

Serialized view

Initially: a = 0

TX_B

```
atomic {
    …
    a = 10 // write a
    …
}
```

TX_A

```
atomic {
    …
    x = a // read a
    if (x == 0) {
        …
    } else {
        …
    }
}
```

Serial order of transactions.

Should have read a == 10
Executions that read a == 0 are invalid!

Transactions can be aborted

Issues like this are handled by a Concurrency Control (CC) mechanism

When a transaction aborts, it can be retried automatically or the user is notified

Bank account example

Initially a = 100; b = 100

TX_A

```
atomic {
        Withdraw(a, 10);
        Deposit(b, 10);
}
```

TX_B

```
atomic {
        Withdraw(b, 5);
        Deposit(a, 5);
}
```

Bank account example

Initially a = 100; b = 100

TX_A

```
atomic {
        Withdraw(a, 10);
        Deposit(b, 10);
}
```

a reads 100,
now local a = 90

TX_B

```
atomic {
        Withdraw(b, 5);
        Deposit(a, 5);
}
```

Bank account example

Initially a = 100; b = 100

TX_A

```
atomic {
        Withdraw(a, 10);
        Deposit(b, 10);
}
```

a reads 100,
now local a = 90

TX_B

```
atomic {
        Withdraw(b, 5);
        Deposit(a, 5);
}
```

b reads 100
now local b = 95

a reads 100
now local a = 105

Bank account example

Initially a = 100; b = 100

TX$_A$

```
atomic {
        Withdraw(a, 10);
        Deposit(b, 10);
}
```

a reads 100,
now local a = 90

TX$_B$

```
atomic {
        Withdraw(b, 5);
        Deposit(a, 5);
}
```

commit
a is now 105
b is now 95

Bank account example

Initially a = 100; b = 100

TX_A

```
atomic {
        Withdraw(a, 10);
        Deposit(b, 10);
}
```

What now?
May b read 95?

TX_B

```
atomic {
        Withdraw(b, 5);
        Deposit(a, 5);
}
```

commit
a is now 105
b is now 95

Zombies and Consistency

Initially a = 10; b = 0; c = 0

TX_A

```
atomic {
        a = a + 10;
        b = b + 10;
        c = 1 / (a-b);
}
```

TX_B

```
atomic {
        a = a - 10;
        b = b + 10;
        c = 1 / (a-b);
}
```

Zombies and Consistency

Initially a = 10; b = 0; c = 0

TX_A

```
atomic {
        a = a + 10;
        b = b + 10;
        c = 1 / (a-b);
}
```

local a = 20

TX_B

```
atomic {
        a = a - 10;
        b = b + 10;
        c = 1 / (a-b);
}
```

commit

a = 0

b = 10

Zombies and Consistency

Initially a = 10; b = 0; c = 0

TX_A

```
atomic {
        a = a + 10;
        b = b + 10;
        c = 1 / (a-b);
}
```

TX_B

```
atomic {
        a = a - 10;
        b = b + 10;
        c = 1 / (a-b);
}
```

if b read 10, we would have a-b = 0 → catastrophic inconsistency

commit
a = 0
b = 10

Consistency Guarantee

The transactional memory system guarantees that *consistent data* will always be seen by a running transaction

Possibilities (conceptually):

- Snapshot at the beginning
- Early abort

Where to implement TM?

Hardware TM (HTM):

- can be fast
- but, bounded resources
- can often not handle big transactions

Examples:

- Intel Haswell → first widely available implementation of TM
- Sun (now Oracle) Rock → was not released
- Supercomputers (IBM's Blue Gene/Q)

Intel Haswell instructions

xbegin: transaction begin

xend: transaction end

xabort: abort transaction

Pattern:

```
xbegin L0
   <transaction code>
xend
   <commit was successful>
   ...
L0:
   <transaction aborted>
```

Where to implement TM?

Software (STM)

- in the (parallel) programming language
- greater flexibility
- achieving good performance might be challenging
- Examples: Haskell, Clojure, …

Hybrid TM (Hardware + Software)

TM is still work in progress!

Implementations still immature

Many different approaches

The first HTM (RTM) implementation just became widely available (Intel Haswell)

STM implementations are still being actively developed

Design choice: strong vs. weak isolation

Q: What happens when shared state accessed by a transaction, is also accessed outside of a transaction?

Are the transactional guarantees still maintained?

Strong isolation: Yes

- easier for porting existing code
- difficult to implement, overhead

Weak isolation: No

Design choice: Nesting

Q: What are the semantics of nested transactions
(Note: nested transactions are important for composability)

- Flat nesting
- Closed nesting
- Other approaches (e.g., open nesting)

Flattened nesting

```
atomic {
  atomic {
    atomic {
      ...
    }
  }
}
```

```
atomic {

      ...

}
```

inner aborts → outer aborts

inner commits → changes visible only if outer commits

Closed nesting

Similar to flattened, but:

- an abort of an inner transaction does not result in an abort for the outer transaction

Inner transaction commits

- changes visible to outer transaction
- but not to other transactions

Outer transaction commits

- changes of inner transactions become visible

What is part of a transaction?

```
stmt1;
stmt2;
stmt3;
```

```
atomic {
  stmt1;
  stmt2;
  stmt3;
}
```

If all program variables are protected:

- easier to port existing code

but, difficult to implement

- need to **check every memory operation**

Reference-based STMs

Mutable state is put into **special variables**

These variables can **only be modified inside a transaction**

Everything else is immutable (or not shared)

This is the model that we will (briefly) discuss

We will use scala-stm

Java does not include STM support

- Scala-stm is an STM library built for scala
- Has a Java interface
- Follows the reference-based (Ref) approach

Other STMs for Java exist (e.g., Deuce), exhibiting a research character
[like also the scala-stm in Java]

scala-stm (on Java) limitations

Java 7 does not have lambdas (Java 8 has!)
→ each transaction is defined as a Runnable Object

No compiler support for ensuring that Refs are only accessed inside a transaction

Our goal is to get a first idea of how to use an STM

- a view of things to come (?)
- not an established programming technique yet

Bank account (scala-stm)

```
class AccountSTM {
  private final Integer id;         // account id
  private final Ref.View<Integer> balance;

   AccountSTM(int id, int balance) {
      this.id      = new Integer(id);
      this.balance = STM.newRef(balance);
  }

}
```

Ideal world: bank account using atomic keyword

```
void withdraw(final int amount) {
      // assume that there are always sufficient funds...
      atomic {
          int old_val = balance.get();
          balance.set(old_val – amount);
      }
}

void deposit(final int amount) {
      atomic {
          int old_val = balance.get();
          balance.set(old_val + amount);
      }
}
```

Real world: bank account in scala-stm

```
void withdraw(final int amount) {
      // assume that there are always sufficient funds...
      STM.atomic(new Runnable() { public void run() {
          int old_val = balance.get();
          balance.set(old_val – amount);
      }});
}

void deposit(final int amount) {
      STM.atomic(new Runnable() { public void run() {
          int old_val = balance.get();
          balance.set(old_val + amount);
      }});
}
```

GetBalance

"atomic"

```
public int getBalance() {
  int result = STM.atomic(
    new Callable<Integer>() {
    public Integer call() {
      int result = balance.get();
      return result;
    }
  });
  return result;
}
```

Bank account transfer

```
static void transfer(final AccountSTM a,
                     final AccountSTM b,
                     final int amount) {
     atomic {
          a.withdraw(amount);
          b.deposit(amount);
     }
}
```

What if account a does not have enough funds?

How can we wait until it does in order to retry the transfer?

locks → conditional variables

TM → retry

Bank account transfer with retry

```
static void transfer_retry(final AccountSTM a,
                           final AccountSTM b,
                           final int amount) {

     atomic {
          if (a.balance.get() < amount)
               STM.retry();
          a.withdraw(amount);
          b.deposit(amount);
     }
}
```

retry: abort the transaction and retry when conditions change

Continue here in the next lesson…

How does retry work?

Implementations need to track what reads/writes a transaction performed to detect conflicts

- Typically called **read-/write-set of a transaction**
- When retry is called, transaction aborts and will be retried when any of the variables that were read, change
- In our example, when a.balance is updated, the transaction will be retried

Question

Does Transactional Memory make concurrent programming always simple?

Dependencies can lead to application level deadlock

Initially x = y = 0

TX_A

```
atomic {
  x = 1;
}
```

$TX_{A'}$

```
atomic {
if (y == 0)
  retry;
}
```

TX_B

```
atomic {
  if (x == 0)
    retry;
  y = 1;
}
```

Simplest STM Implementation

Ingredients

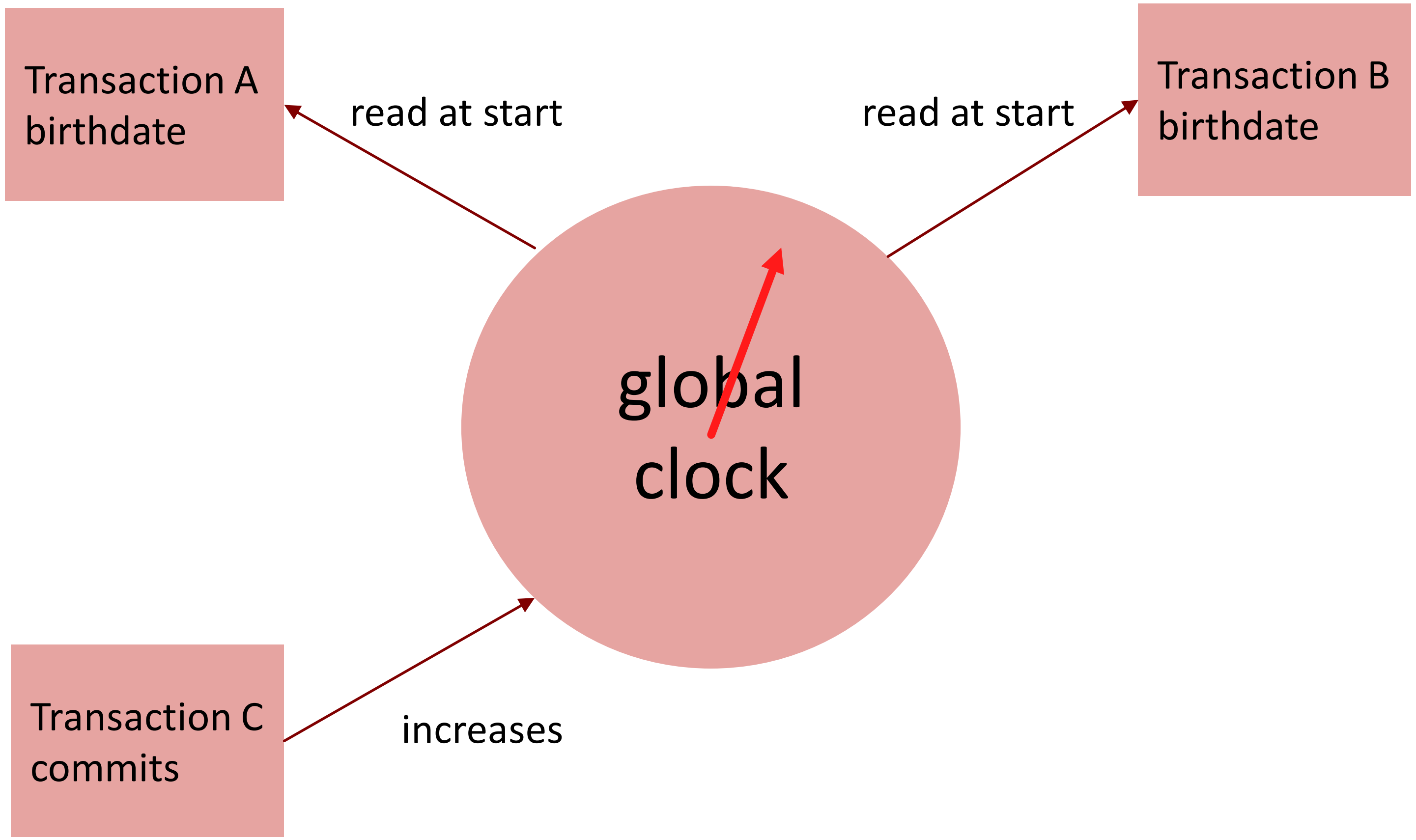
Threads that run transactions with thread states

- active
- aborted
- committed

Objects representing state stored in memory (the variables affected by a transaction)

- offering methods like a constructor, read, write
- and copy!

Clock-based STM System



Atomic Objects

Each transaction uses a local **read-set** and a local **write-set** holding all locally read and written objects.

atomic memory object	
version reference	time stamp

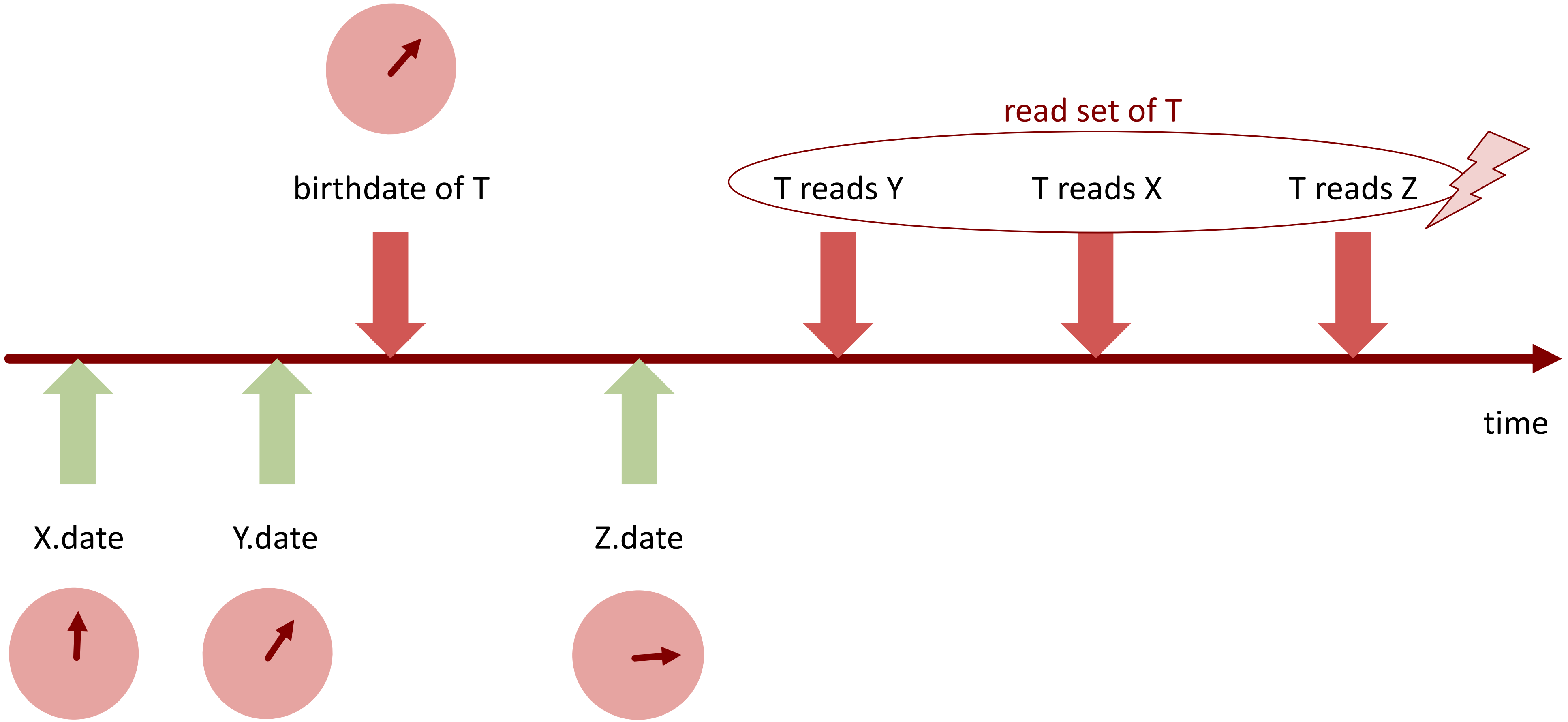
Transaction calls **read**

- check if the object is in the write set → return this (new) version
- otherwise check if object's time stamp <= transaction's birthdate, if not throw aborted exception, otherwise add new copy of the object to the read set

Transaction calls **write**

- if object is not in write set, create a copy of it in the write set

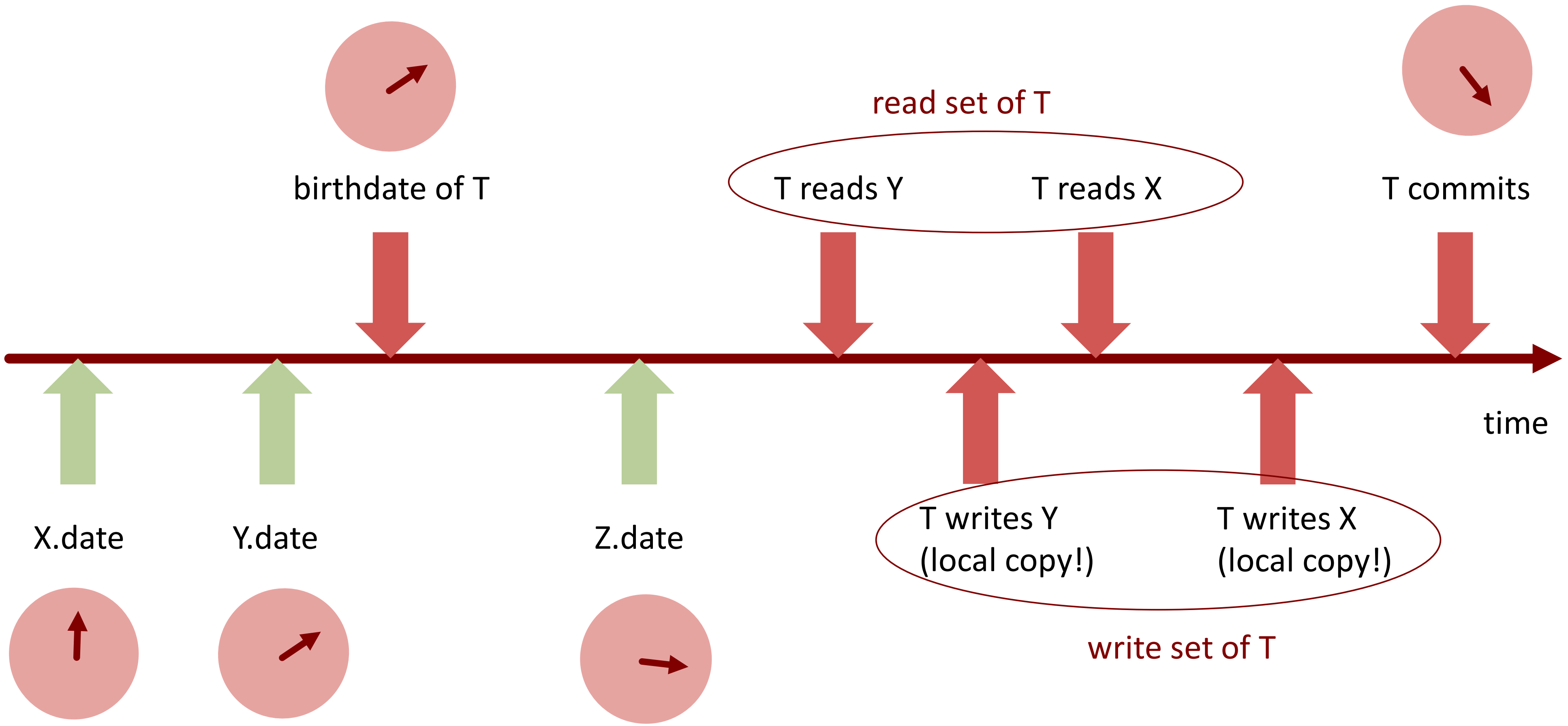
Transaction life time



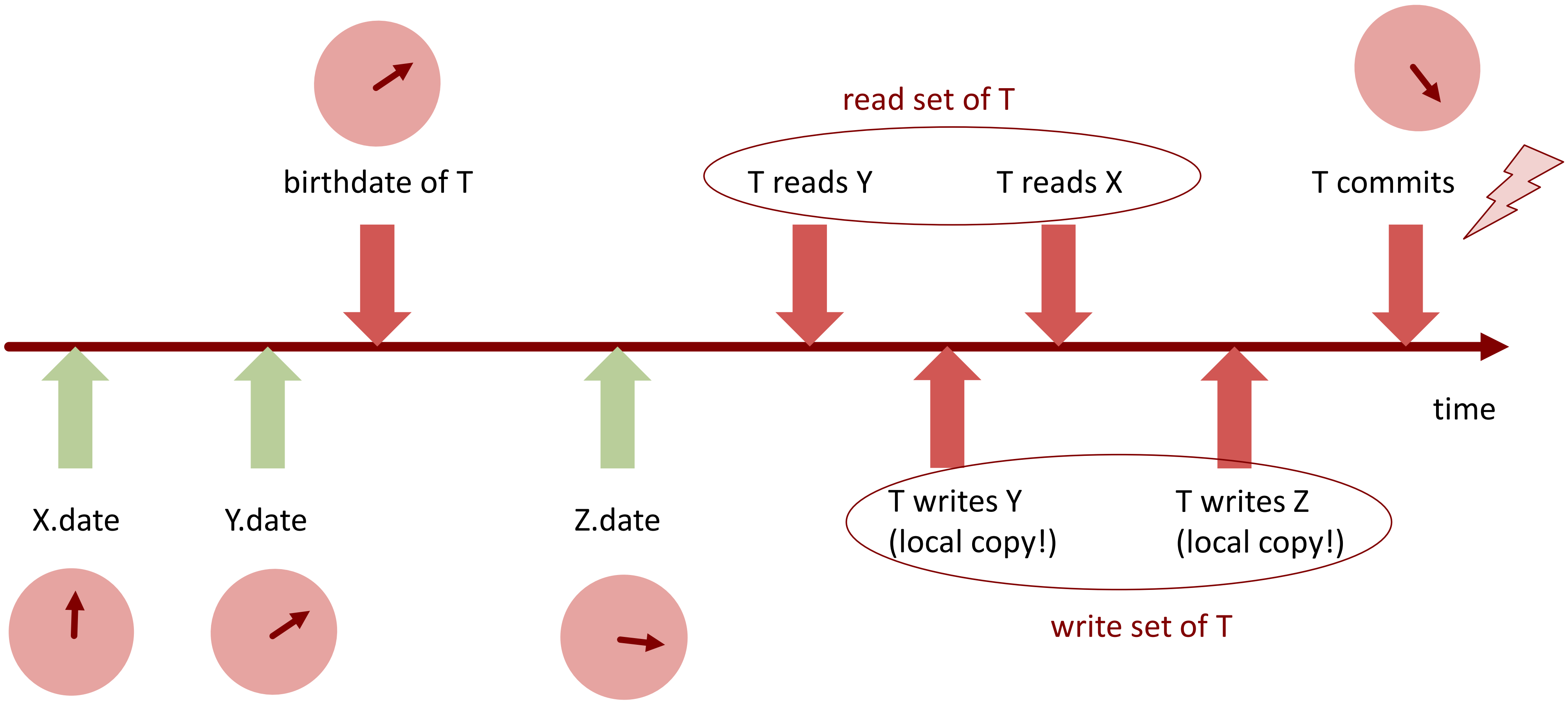
Commit

- Lock all objects of read- and write-set (in some defined order to avoid deadlocks)
- Check that all objects in the read set provide a time stamp <= birthdate of the transaction, otherwise return "abort"
- Increment and get the value T of current global clock
- Copy each element of the write set back to global memory with timestamp T
- Release all locks and return "commit"

Successful commit



Aborted commit



Dining philosophers

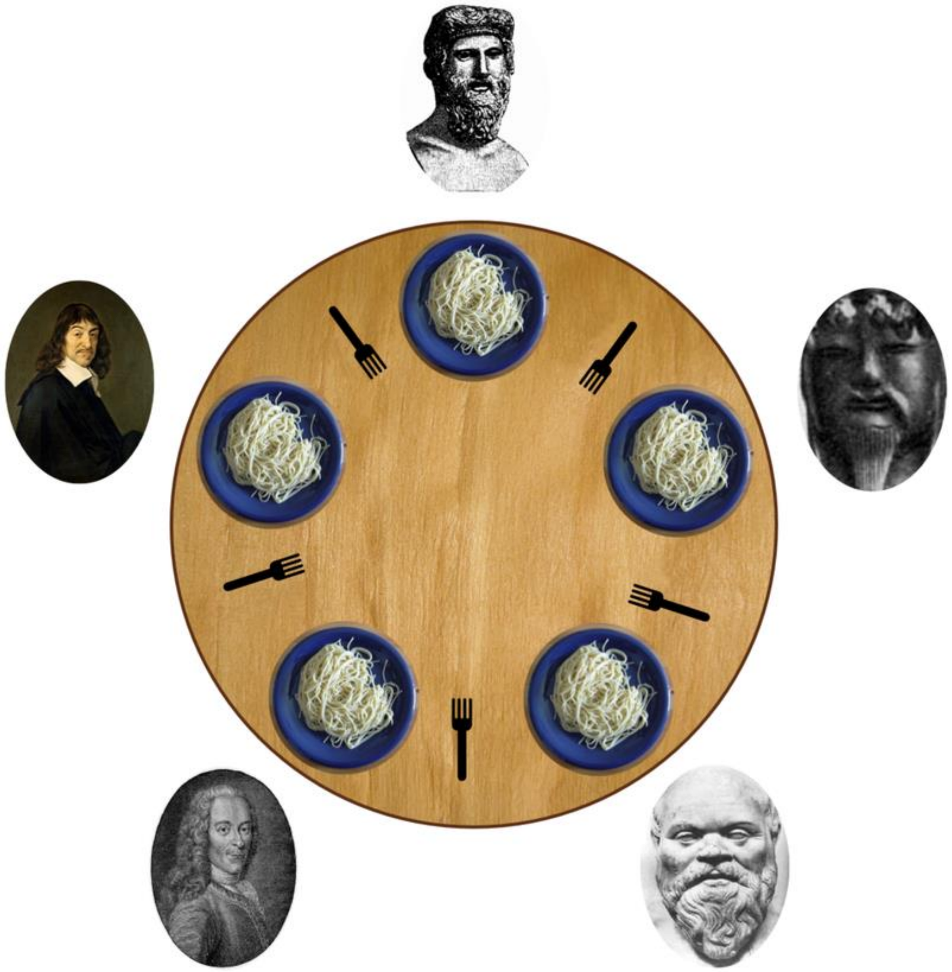
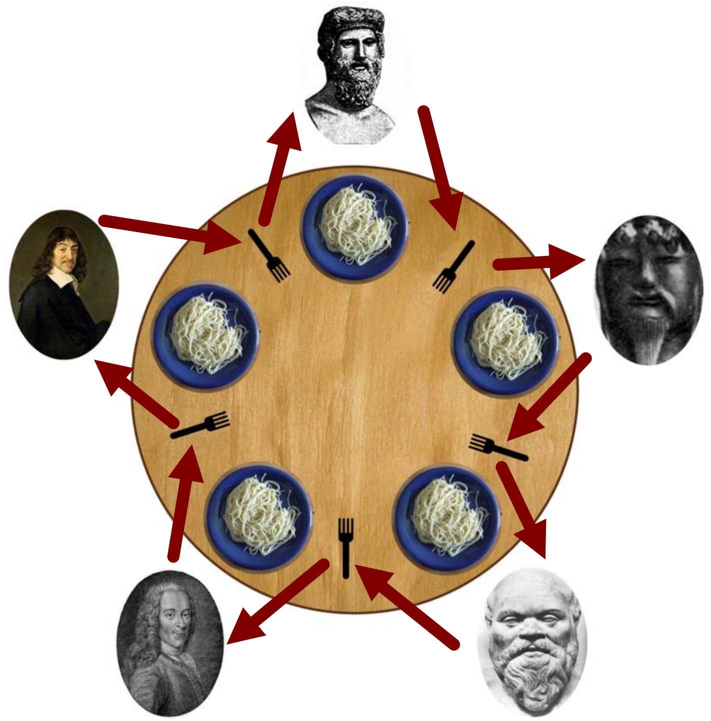


image source: Wikipedia

- 5 philosophers
- 5 forks
- each philosopher requires 2 forks to eat
- forks cannot be shared

Solution that can lead to deadlock

Philosopher:
- think
- lock left
- lock right
- eat
- unlock right
- unlock left



P_1 takes F_1, P_2 takes F_2, P_3 takes F_3, P_4 takes F_4, P_5 takes F_5
→ Deadlock

Dining Philosophers Using TM

```
private static class Fork {
      public final Ref.View<Boolean> inUse = STM.newRef(false);
}
class PhilosopherThread extends Thread {
      private final int meals;
      private final Fork left;
      private final Fork right;

      public PhilosopherThread(Fork left, Fork right) {
            this.left = left;
            this.right = right;
      }

      public void run() { … }
}
```

Dining Philosophers Using TM

```
Fork[] forks = new Fork[tableSize];

for (int i = 0; i < tableSize; i++)
      forks[i] = new Fork();


PhilosopherThread[] threads =
      new PhilosopherThread[tableSize];

for (int i = 0; i < tableSize; i++)
      threads[i] = new PhilosopherThread(
                          forks[i],
                       forks[(i + 1) % tableSize]);
```

Dining Philosophers Using TM

```
class PhilosopherThread extends Thread {
    …
        public void run() {
            for (int m = 0; m < meals; m++) {
                // THINK
                pickUpBothForks();
                // EAT
                putDownForks();
            }
        }
    …
}
```

Dining Philosophers Using TM

```
class PhilosopherThread extends Thread {
    …
        private void pickUpBothForks() {
                STM.atomic(new Runnable() { public void run() {

                    if (left.inUse.get() || right.inUse.get())
                        STM.retry();

                    left.inUse.set(true);
                    right.inUse.set(true);

                }});
        }
    …
}
```

Dining Philosophers Using TM

```
class PhilosopherThread extends Thread {
    …
        private void putDownForks() {
            STM.atomic(new Runnable() { public void run() {

                    left.inUse.set(false);
                    right.inUse.set(false);

            }});
        }
    …
}
```

Issues with transactions

- It is not clear what are the best semantics for transactions
- Getting good performance can be challenging
- I/O operations (e.g., print to screen)
  Can we perform I/O operations in a transaction?

Summary

- Locks are too hard!
- Transactional Memory tries to remove the burden from the programmer
- STM / HTM

- Remains to be seen whether it will be widely adopted in the future

Additional Reading

Simon Peyton Jones,
Beautiful concurrency
http://research.microsoft.com/pubs/74063/beautiful.pdf

Dan Grossman,
The Transactional Memory / Garbage Collection Analogy
https://homes.cs.washington.edu/~djg/papers/analogy_oopsla07.pdf

Distributed Memory & Message Passing

So far

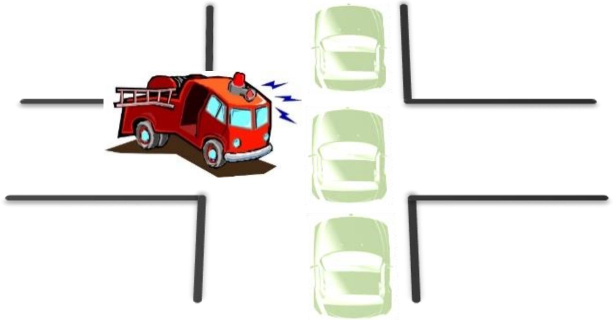
Considered

- Parallel / Concurrent
- Fork-Join / Threads
- OOP on Shared Memory
- Locking / Lock Free / Transactional
- Semaphores / Monitors

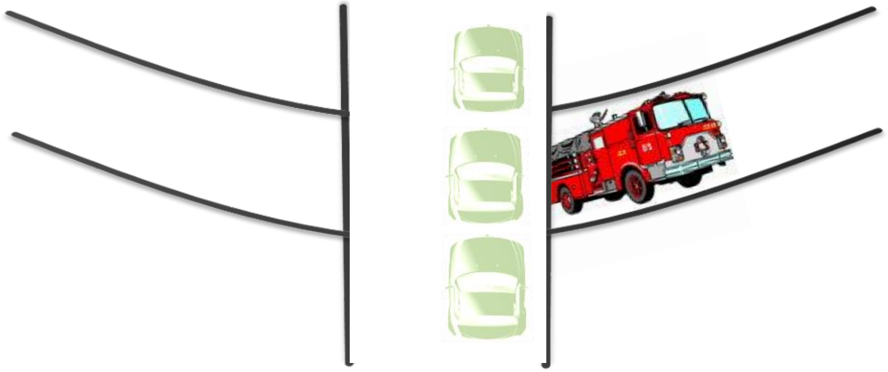
Sharing State

Many of the problems of parallel/concurrent programming come from sharing state

- Complexity of locks, race conditions, ….



What if we avoid sharing state?



Alternatives

Functional Programming

- Immutable state → no synchronization required

Message Passing: Isolated mutable state

- State is mutable, but not shared: Each thread/task has its private state
- Tasks cooperate via message passing

Concurrent Message Passing

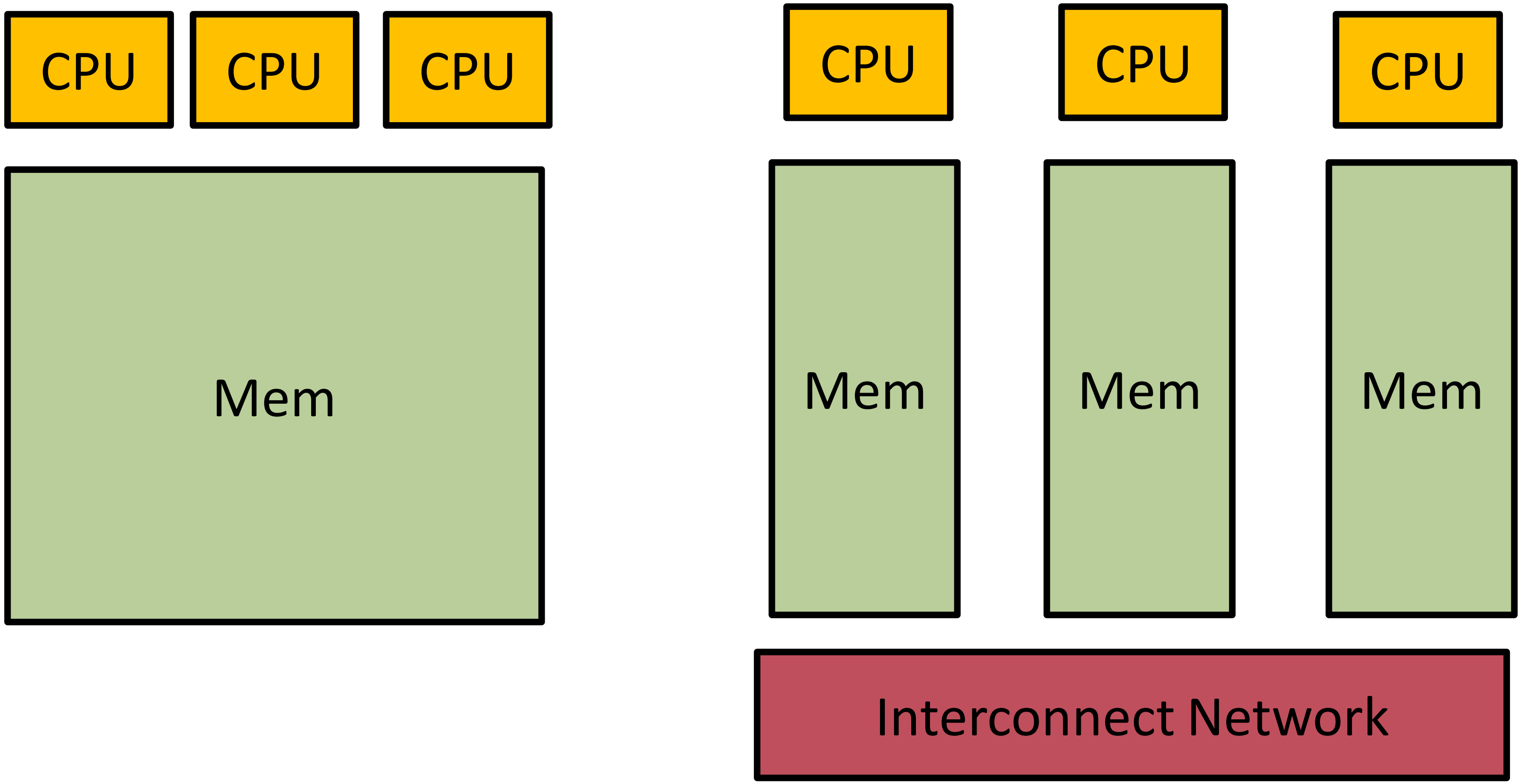
Programming Models

- CSP: Communicating Sequential Processes
- Actor programming model

Framework

- MPI (Message Passing Interface)

Shared vs Distributed memory



Shared/Distributed memory programming models

Shared memory architectures (e.g., multicores)

- Both message passing and sharing state is used
- Message passing in shared memory:
  - Can be slower than sharing data
  - Easy to implement
  - Arguably, easier to reason about

Distributed memory architectures (e.g., datacenters, supercomputers, clusters)

- Sharing state is challenging and often inefficient
- Almost exclusively use message passing (slowly changing though)
- Additional concerns: e.g., failures