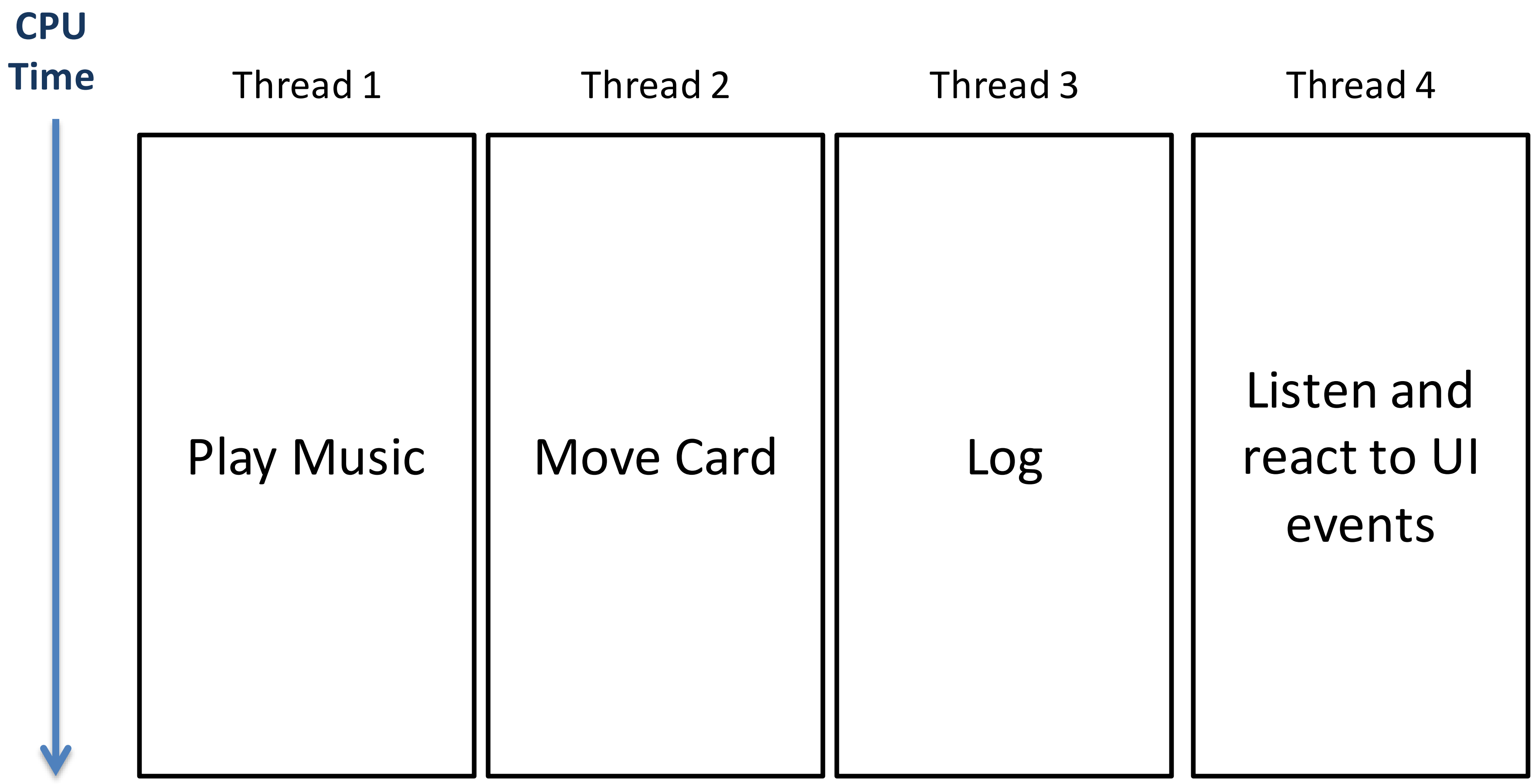
Today: Making Threads Play Nice (and Share Their Data)

1. A weekday at the data races spoiled by dangerous non-atomic weapons.
2. Locking an object so one leaves with it.
3. Too many locks make a dead one.
4. Conditional release for good behavior.
5. Synchronizers and other thread-safe classes

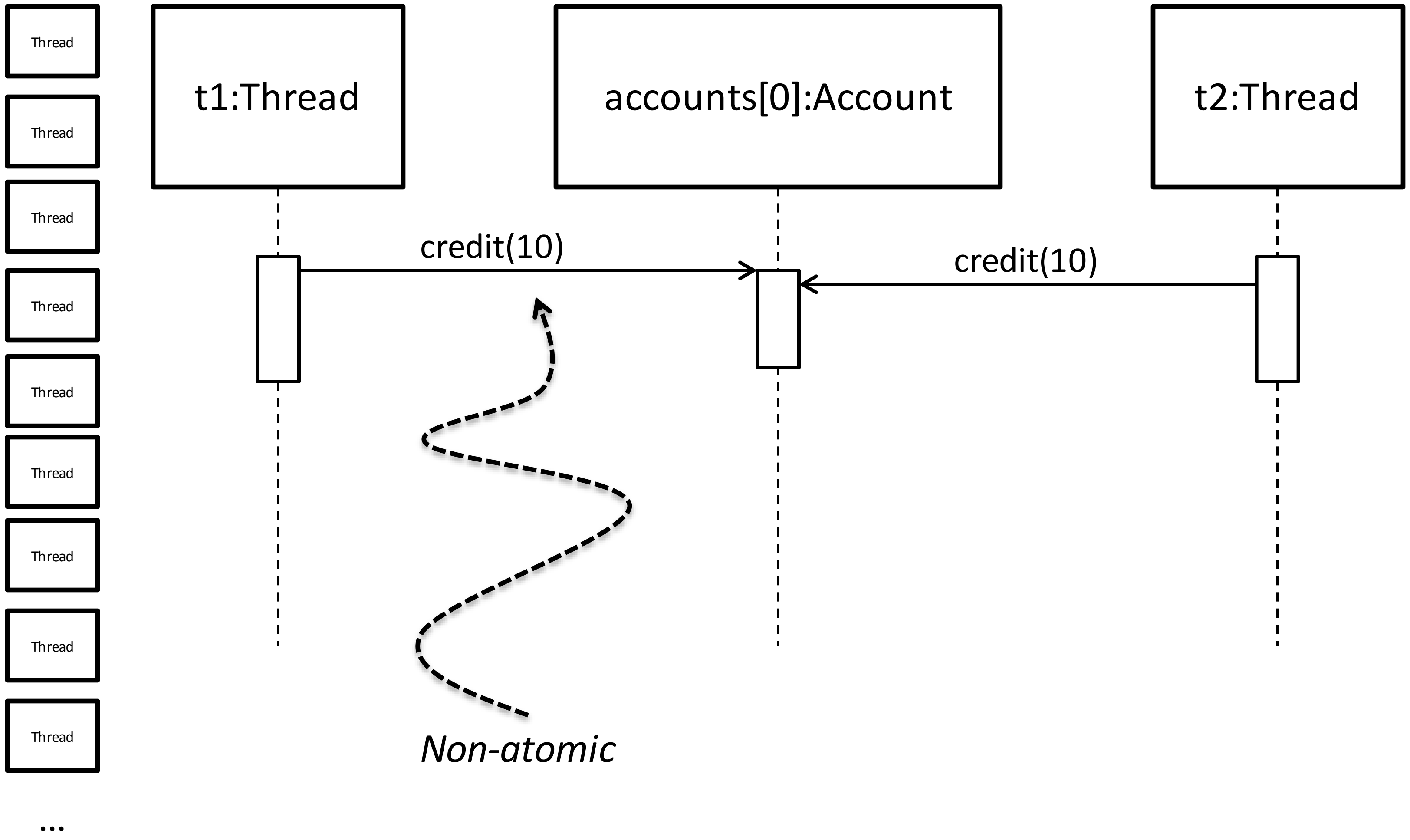
What’s The Simplifying Assumption Here?



Next Week

- Generics (Lucas)
- TaskNav (Marc & Mathieu)
  - Applying COMP 303 material to large(r)-scale development
  - Web application development with Spring Framework, MongoDB, and Solr
  - The summer internship experience at McGill
  - The FacSci Undergraduate Research Competition

Contention on an Account object



Credit Play-by-Play 1

T1 Code	R1	Accounts[0]	R2	T2 Code
Load	20000	20000	-	<wait>
Add 10	20010	20000	-	<wait>
Store	20010	20010	-	<wait>
<wait>	20010	20010	20010	Load
wait	20010		20020	Add 10
wait	20010	20020	20020	Store

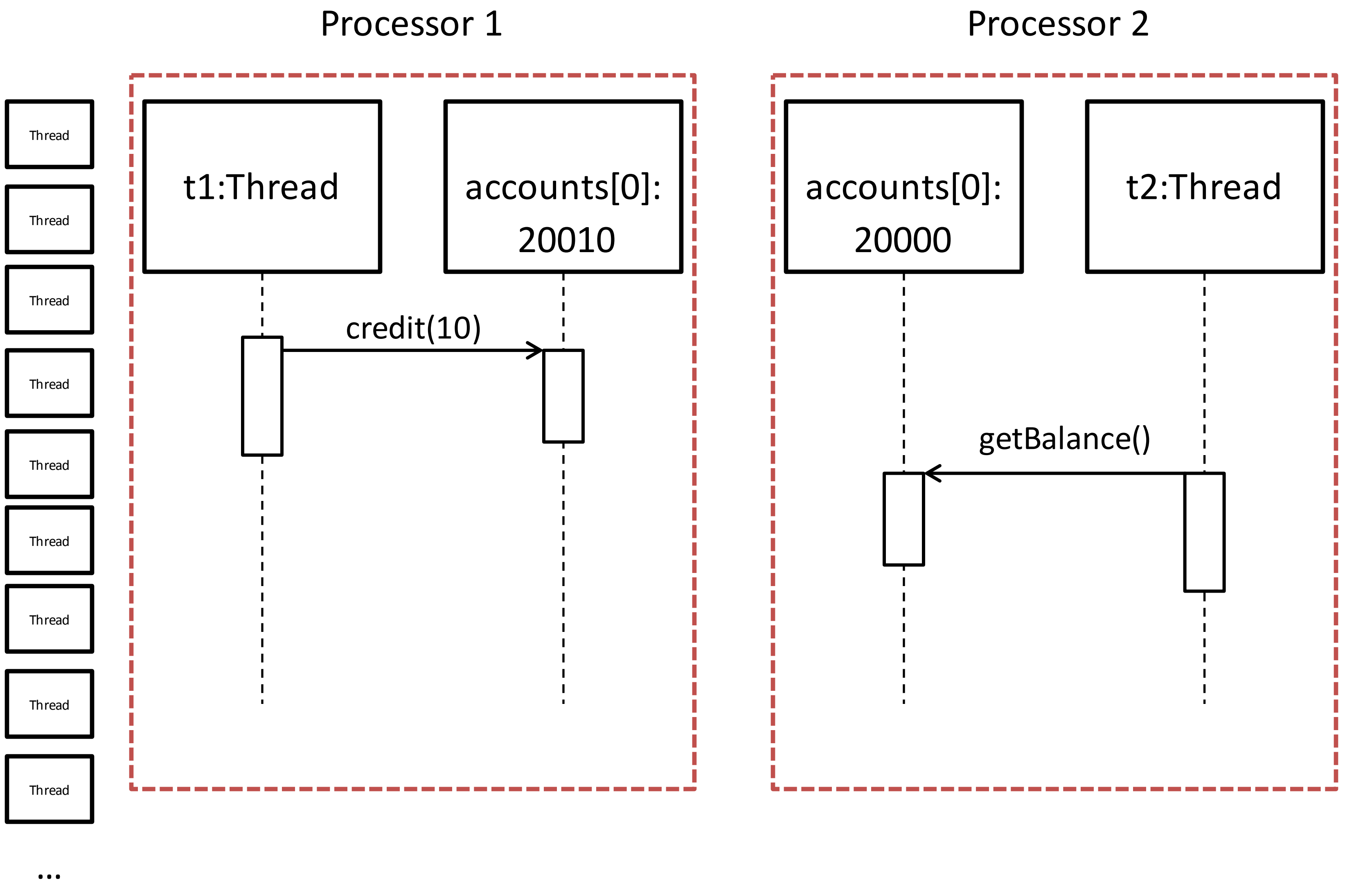
Credit Play-by-Play 2

T1 Code	R1	Accounts[0]	R2	T2 Code
Load	20000	20000		<wait>
Add 10	20010	20000		<wait>
<wait>	20010	20000	20000	Load
wait	20010	20000	20010	Add 10
wait	20010	20010	20010	Store
Store	20010	20010	20010	<wait>

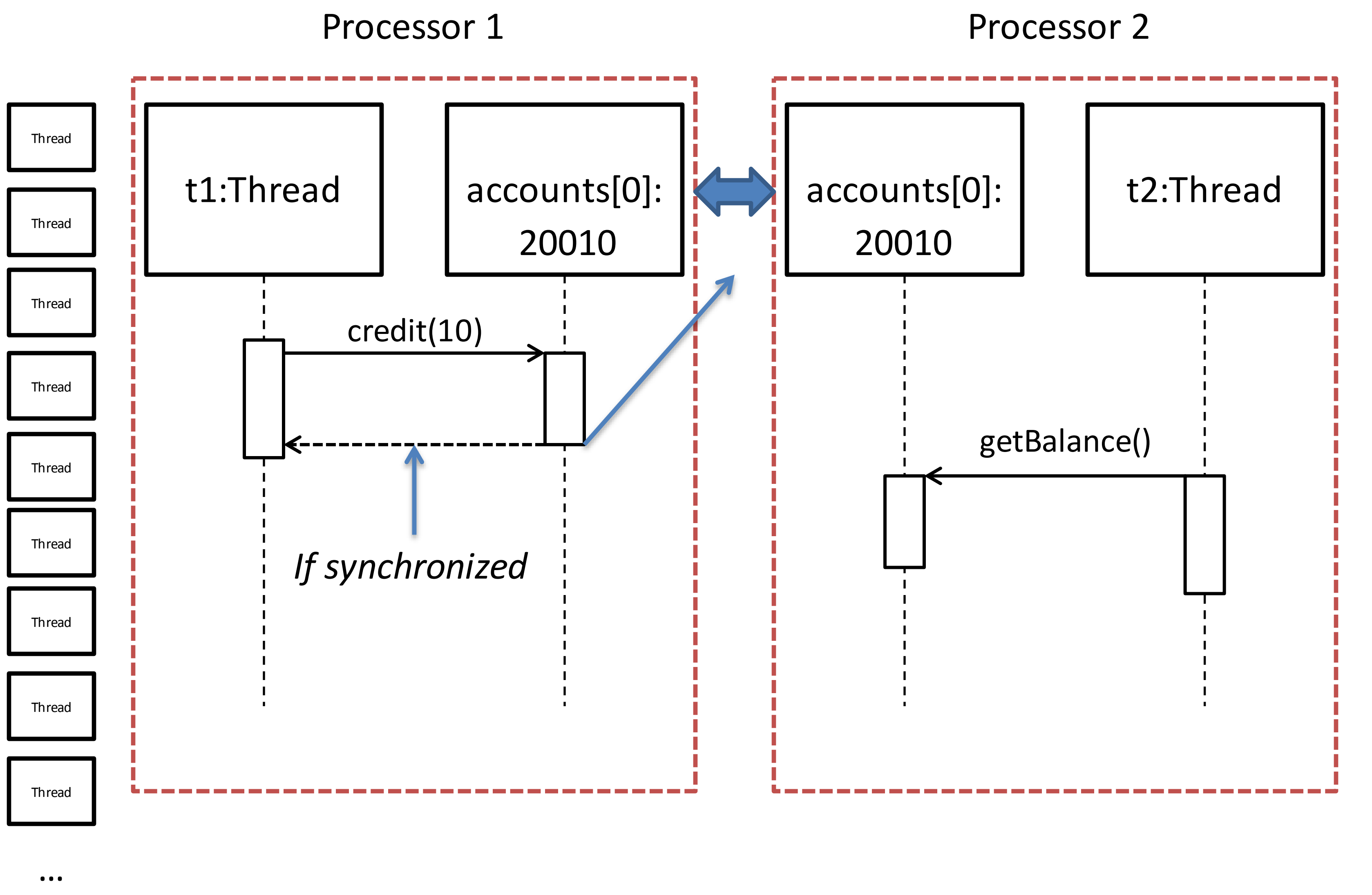
Credit Play-by-Play 2

T1 Code	R1	L	Accounts[0]	R2	T2 Code
acquire		1	20000		<w>
Load	20000	1	20000		<w>
Add 10	20010	1	20000		<w>
					acquire
Store	20010	1	20010		
Release					
		2			acquire

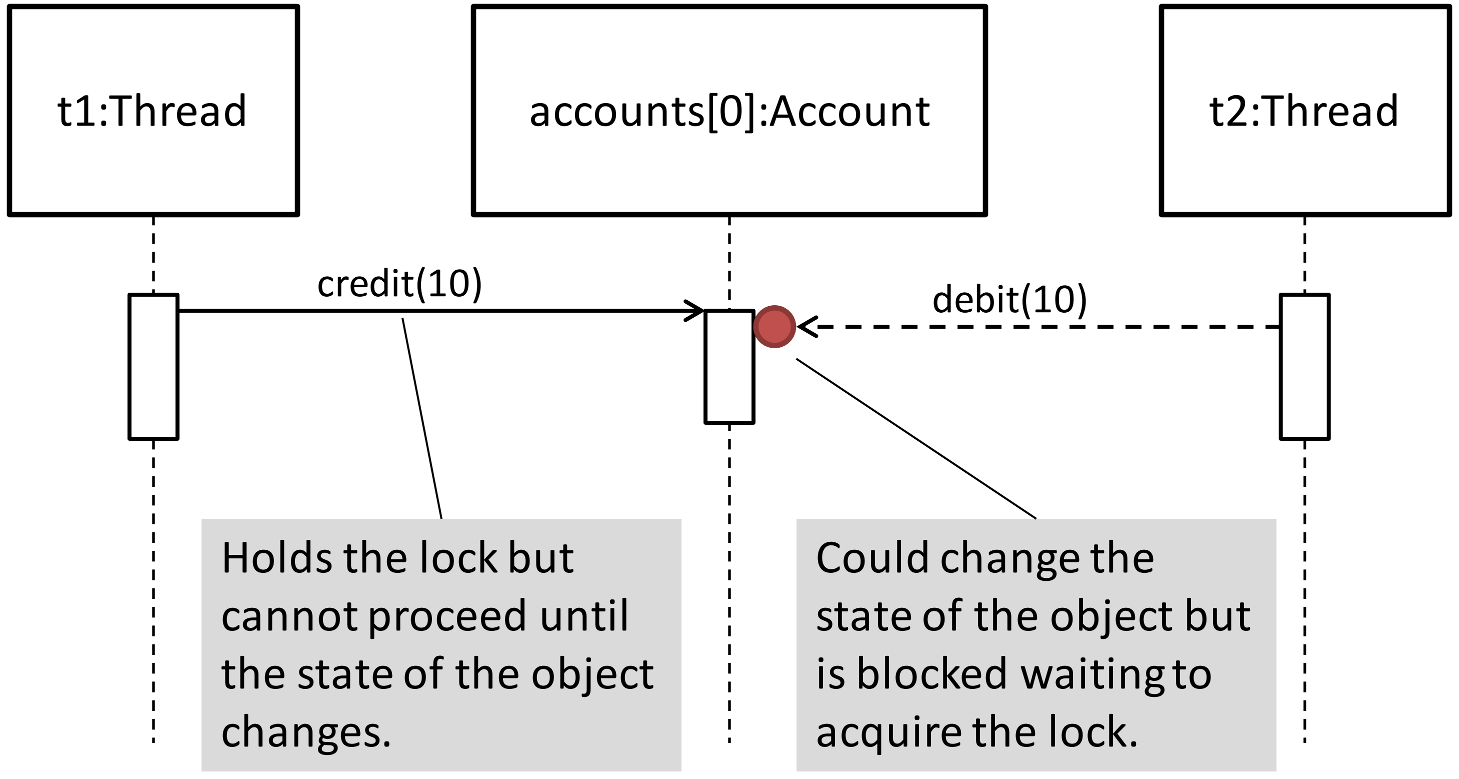
A More Complex View of Shared State



A More Complex View of Shared State



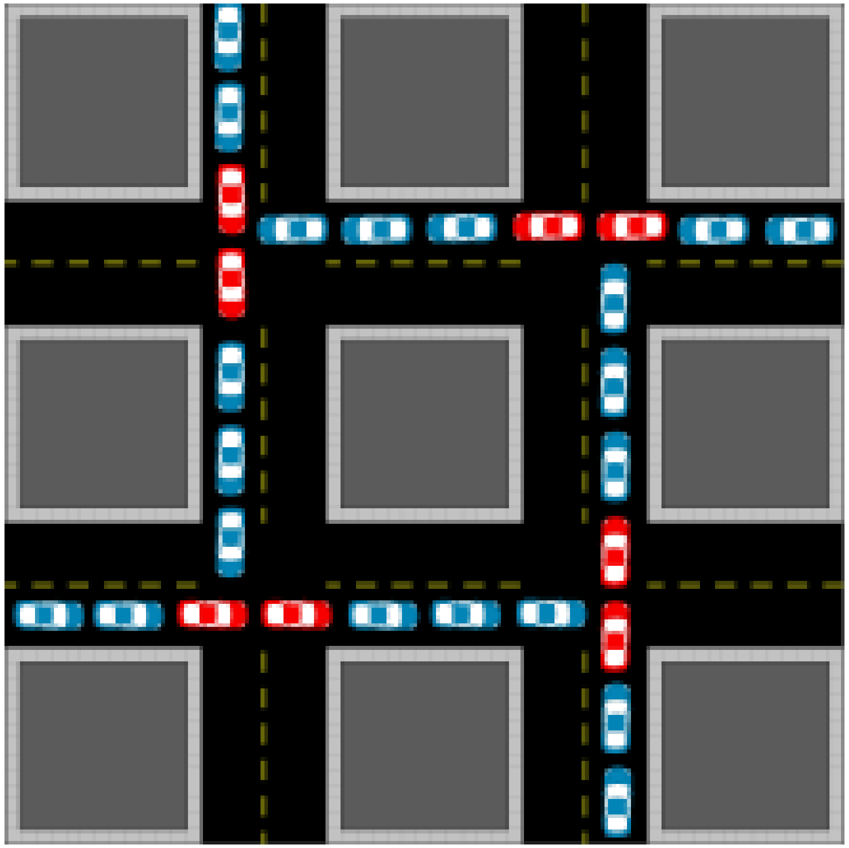
Deadlock



Holds the lock but cannot proceed until the state of the object changes.

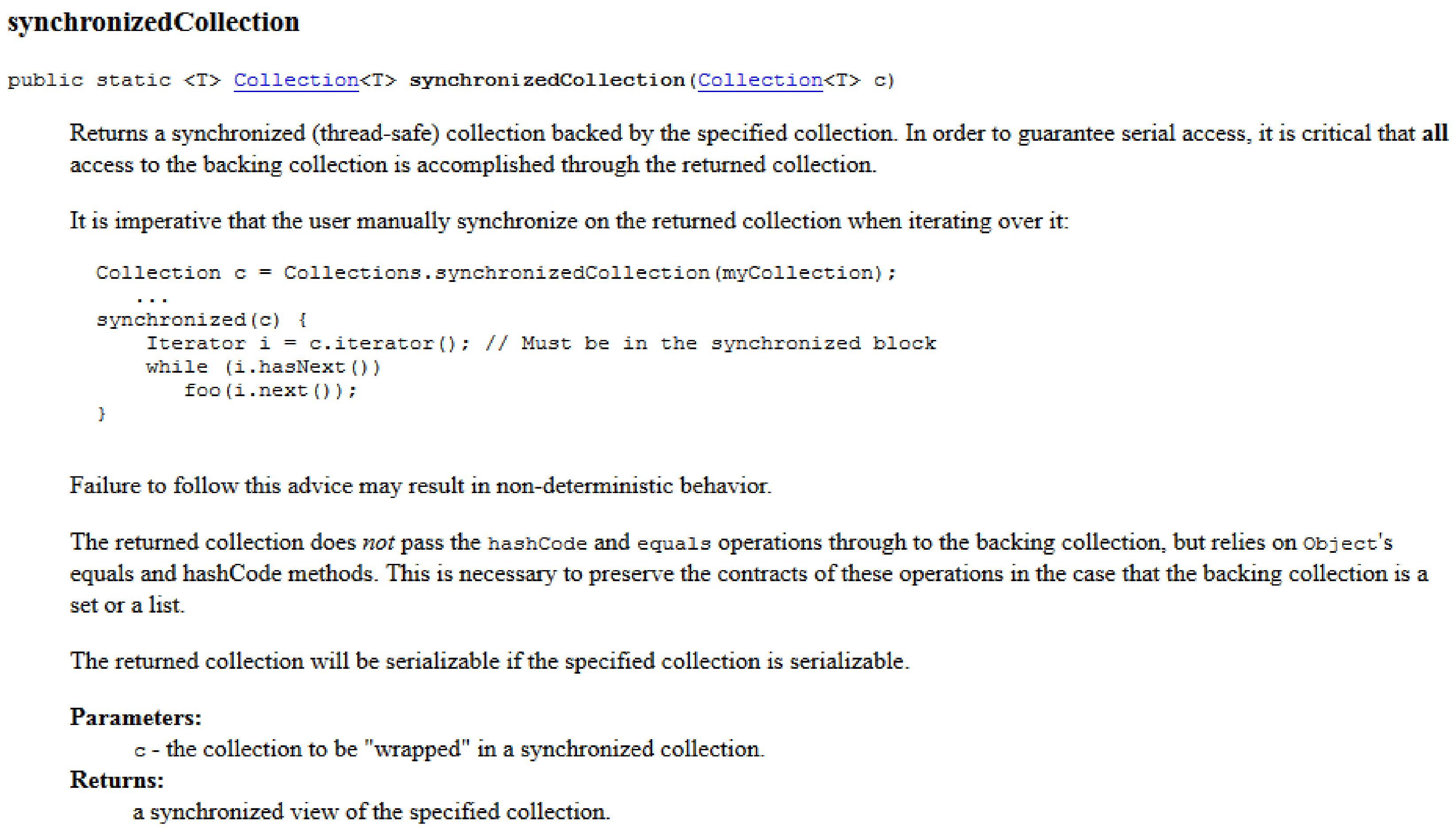
Could change the state of the object but is blocked waiting to acquire the lock.

Gridlock



Source: https://en.wikipedia.org/wiki/Gridlock]

Important Concurrency APIs



synchronizedCollection

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c)
```

Returns a synchronized (thread-safe) collection backed by the specified collection. In order to guarantee serial access, it is critical that **all** access to the backing collection is accomplished through the returned collection.

It is imperative that the user manually synchronize on the returned collection when iterating over it:

```
Collection c = Collections.synchronizedCollection(myCollection);
   ...
synchronized(c) {
    Iterator i = c.iterator(); // Must be in the synchronized block
    while (i.hasNext())
       foo(i.next());
}
```

Failure to follow this advice may result in non-deterministic behavior.

The returned collection does *not* pass the `hashCode` and `equals` operations through to the backing collection, but relies on `Object`'s equals and hashCode methods. This is necessary to preserve the contracts of these operations in the case that the backing collection is a set or a list.

The returned collection will be serializable if the specified collection is serializable.

Parameters:

`c` - the collection to be "wrapped" in a synchronized collection.

Returns:

a synchronized view of the specified collection.

Package java.util.concurrent

Utility classes commonly useful in concurrent programming.

See:
   Description

Interface Summary	
BlockingDeque<E>	A Deque that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.
BlockingQueue<E>	A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
Callable<V>	A task that returns a result and may throw an exception.
CompletionService<V>	A service that decouples the production of new asynchronous tasks from the consumption of the results of completed tasks.
ConcurrentMap<K,V>	A Map providing additional atomic `putIfAbsent`, `remove`, and `replace` methods.
ConcurrentNavigableMap<K,V>	A ConcurrentMap supporting NavigableMap operations, and recursively so for its navigable sub-maps.
Delayed	A mix-in style interface for marking objects that should be acted upon after a given delay.
Executor	An object that executes submitted Runnable tasks.
ExecutorService	An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.
Future<V>	A `Future` represents the result of an asynchronous computation.

Review: Another Concurrency Bug

```
public class Clock2
  extends java.applet.Applet
  implements Runnable {
...
Thread timer = null;
...
public void start() {
  if (timer == null) {
      timer = new Thread(this);
      timer.start();
    }
}
public void stop() {
  timer = null;
}
public void run() {
  while (timer != null) {
    try {
      Thread.sleep(100);
    } catch (InterruptedException e){}
    repaint();
  }
  timer = null;
}
```