CONCURRENCY

F. Tip and
M. Weintraub

SHARED-MEMORY CONCURRENCY

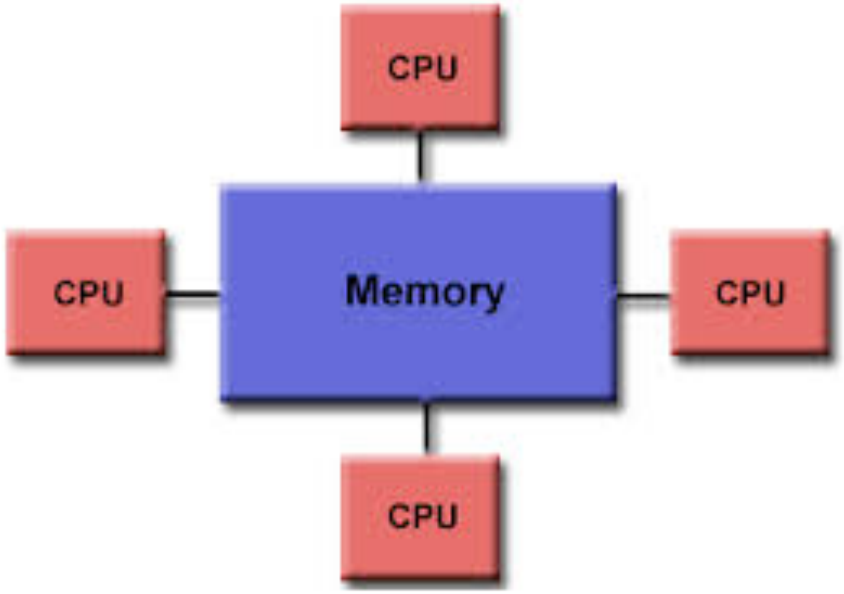
- threads execute **concurrently**
- threads communicate values via **shared memory**
- **synchronization** using locks

PITFALLS

- data races
- atomicity violations
- deadlock

SHARED MEMORY

- memory is accessed simultaneously by multiple threads/CPUs/cores
- no explicit communication operations -- just read/write shared locations
- synchronization operations for concurrency control



JAVA THREADS

```
class Counter {
  public void add(long value) {
    long temp = count + value;
    count = temp;
  }
  public void report(){
    System.out.println("count = " + count);
  }
  private long count = 0;
}
```

- simple Counter with add() and report() methods
- now let’s assume we want to have multiple threads concurrently adding to the counter…

JAVA THREADS

```java
public class Adder implements Runnable {
  Adder(Counter counter, int value){
    this.value = value;
    this.counter = counter;
  }
  public void run() {
    counter.add(value);
  }
  private Counter counter;
  private int value;
}
```

- define a class that implements java.lang.Runnable with a run() method containing the code we want to execute concurrently with other threads

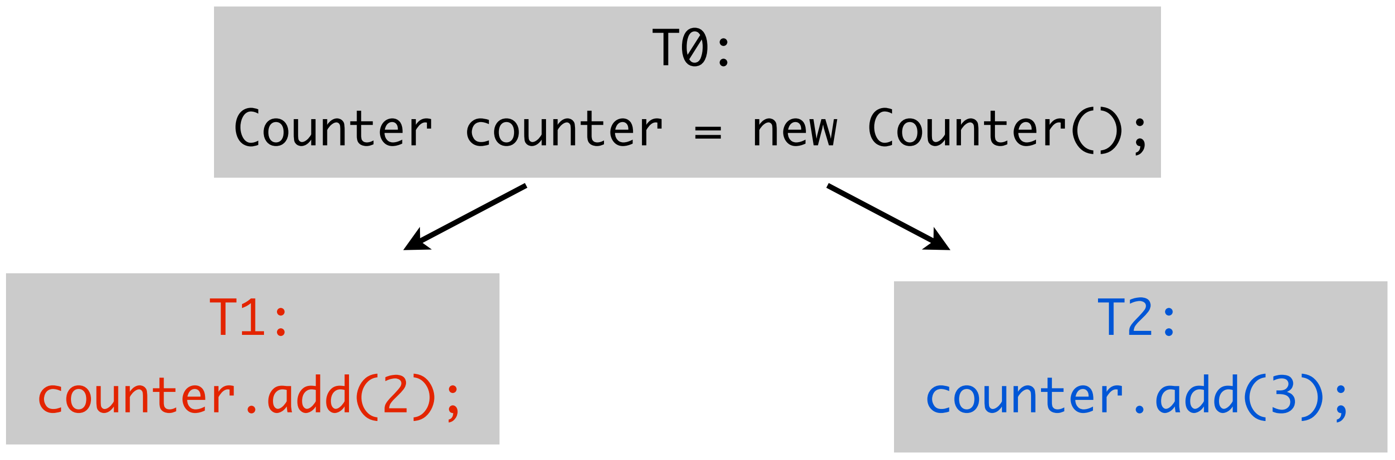
JAVA THREADS

```
public class Example {
  public static void main(String[] args){
    Counter counter = new Counter();
    Adder add2 = new Adder(counter, 2);
    Adder add3 = new Adder(counter, 3);
    Thread thread1 = new Thread(add2);
    Thread thread2 = new Thread(add3);
    thread1.start();
    thread2.start();

    try {
      thread1.join();
      thread2.join();
      counter.report();
    } catch (InterruptedException e) {
      System.err.println("an error occurred");
    }

  }
}
```

- create new Threads, call Thread.start on them()
- invoke Thread.join() to wait for another thread to finish

SUPPOSE TWO THREADS CONCURRENTLY ACCESS A COUNTER..

```
class Counter {
  public void add(long value) {
    long temp = count + value;
    count = temp;
  }
  public void report(){
    System.out.println("count = " + count);
  }
  private long count = 0;
}
```



```
T0:
Counter counter = new Counter();
```

```
T1:
counter.add(2);
```

```
T2:
counter.add(3);
```

SCHEDULE 1: T1 BEFORE T2

```
count = 0;
temp = count + value;          // temp = 0 + 2
count = temp;                  // count = 2
temp = count + value;          // temp = 2 + 3
count = temp;                  // count = 5
```

SCHEDULE 1: T1 BEFORE T2

```
count = 0;
temp = count + value;        // temp = 0 + 2
count = temp;                // count = 2
temp = count + value;        // temp = 2 + 3
count = temp;                // count = 5
```

5

SCHEDULE 2: T2 BEFORE T1

```
count = 0;
temp = count + value;        // temp = 0 + 3
count = temp;                // count = 3
temp = count + value;        // temp = 3 + 2
count = temp;                // count = 5
```

SCHEDULE 2: T2 BEFORE T1

```
count = 0;
temp = count + value;          // temp = 0 + 3
count = temp;                  // count = 3
temp = count + value;          // temp = 3 + 2
count = temp;                  // count = 5
```

5

SCHEDULE 3

```
count = 0;
temp = count + value;        // temp = 0 + 2
temp = count + value;        // temp = 0 + 3
count = temp;                // count = 2
count = temp;                // count = 3
```

SCHEDULE 3

```
count = 0;
temp = count + value;        // temp = 0 + 2
temp = count + value;        // temp = 0 + 3
count = temp;                // count = 2
count = temp;                // count = 3
```

3

SCHEDULE 3

```
count = 0;
temp = count + value;          // temp = 0 + 2
temp = count + value;          // temp = 0 + 3
count = temp;                  // count = 2
count = temp;                  // count = 3
```

3

- problem: the calls to add() are not executed atomically
- **data race**: two threads concurrently access a shared location, and at least one of them is a write, and no synchronization exists between the threads

CONCURRENCY CONTROL

- the result of a concurrent computation depends on the order in which threads are scheduled

- some schedules produce undesirable results

- use synchronization (locks) to prevent undesirable thread schedules

PREVENTING DATA RACES USING LOCKS

```
class Counter {
  public void add(long value) {
    synchronized (lock){
      long temp = count + value;
      count = temp;
    }
  }
  public void report(){
    System.out.println("count = " + count);
  }
  private long count = 0;
  Object lock = new Object();
}
```

obtain lock, to prevent undesirable schedule 3 from happening

- only one thread at a time can enter the region protected by the lock
- any object can be used as a lock

PREVENTING DATA RACES USING LOCKS

```
class Counter {
  public void add(long value) {
    synchronized (this){
      long temp = count + value;
      count = temp;
    }
  }
  public void report(){
    System.out.println("count = " + count);
  }
  private long count = 0;
}
```

- lock on the object on which the method was invoked

PREVENTING DATA RACES USING LOCKS

```
class Counter {
  public synchronized void add(long value) {
    long temp = count + value;
    count = temp;
  }
  public void report(){
    System.out.println("count = " + count);
  }
  private long count = 0;
}
```

- special syntax for a method in which the entire body is protected by a lock on this

PROBLEM SOLVED?

reported in [Wang & Stoller PPoPP’06]

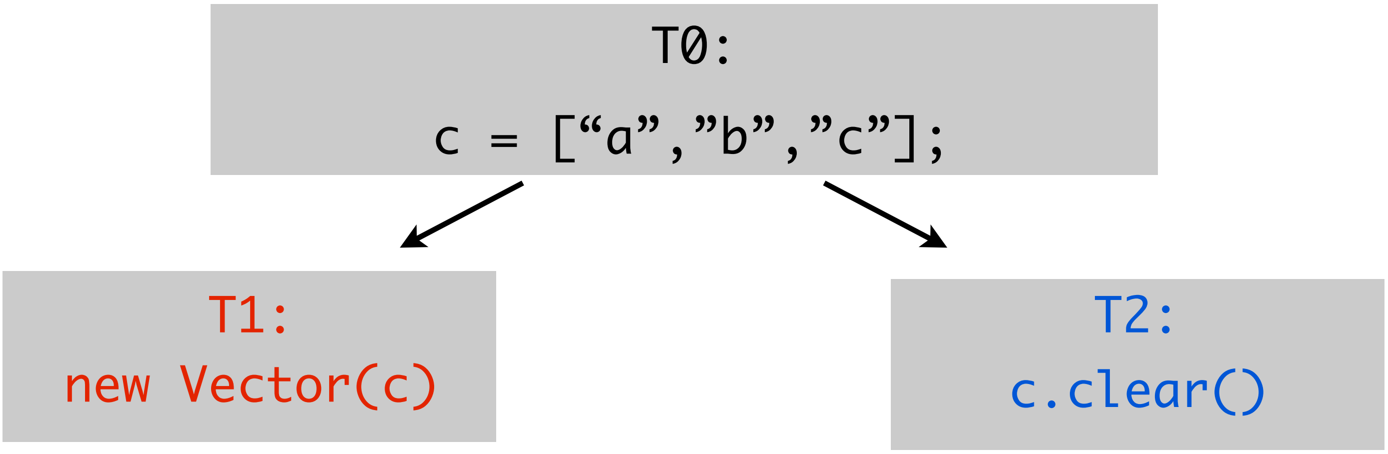
JAVA.UTIL.VECTOR

```
class Vector extends ... implements ... {
  public Vector(Collection c){
    elementCount = c.size();
    elementData = new Object[(int)Math.min(
           (elementCount*110L)/100,Integer.MAX_VALUE)];
    c.toArray(elementData);
  }
  ...
}
```

reported in [Wang & Stoller PPoPP’06]

JAVA.UTIL.VECTOR

```
class Vector extends ... implements ... {
  public Vector(Collection c){
    elementCount = c.size();
    elementData = new Object[(int)Math.min(
           (elementCount*110L)/100,Integer.MAX_VALUE)];
    c.toArray(elementData);
  }
  ...
}
```



T0:

c = [“a”,”b”,”c”];

T1:

new Vector(c)

T2:

c.clear()

VECTOR

```
         class Vector extends ... implements ... {
T1  →      public Vector(Collection c){
             elementCount = c.size();
             elementData = new Object[(int)Math.min(
                    (elementCount*110L)/100,Integer.MAX_VALUE)];
             c.toArray(elementData);
           }

         }
```

```
elementCount = 0
elementData = null
c = [“a”, “b”, “c”]
```

VECTOR

```
        class Vector extends ... implements ... {
          public Vector(Collection c){
            elementCount = c.size();
T1 ⟶        elementData = new Object[(int)Math.min(
                   (elementCount*110L)/100,Integer.MAX_VALUE)];
            c.toArray(elementData);
          }

        }
```

```
elementCount = 3
elementData = null
c = [“a”, “b”, “c”]
```

VECTOR

```
        class Vector extends ... implements ... {
          public Vector(Collection c){
            elementCount = c.size();
            elementData = new Object[(int)Math.min(
                   (elementCount*110L)/100,Integer.MAX_VALUE)];
T1 →        c.toArray(elementData);
          }

        }
```

```
elementCount = 3
elementData = [null, null, null]
c = [“a”, “b”, “c”]
```

VECTOR

```
           class Vector extends ... implements ... {
             public Vector(Collection c){
               elementCount = c.size();
               elementData = new Object[(int)Math.min(
                          (elementCount*110L)/100,Integer.MAX_VALUE)];
 T2 ——→        c.toArray(elementData);
             }

           }
```

```
T2   c.clear()
```

```
elementCount = 3
elementData = [null, null, null]
c = []
```

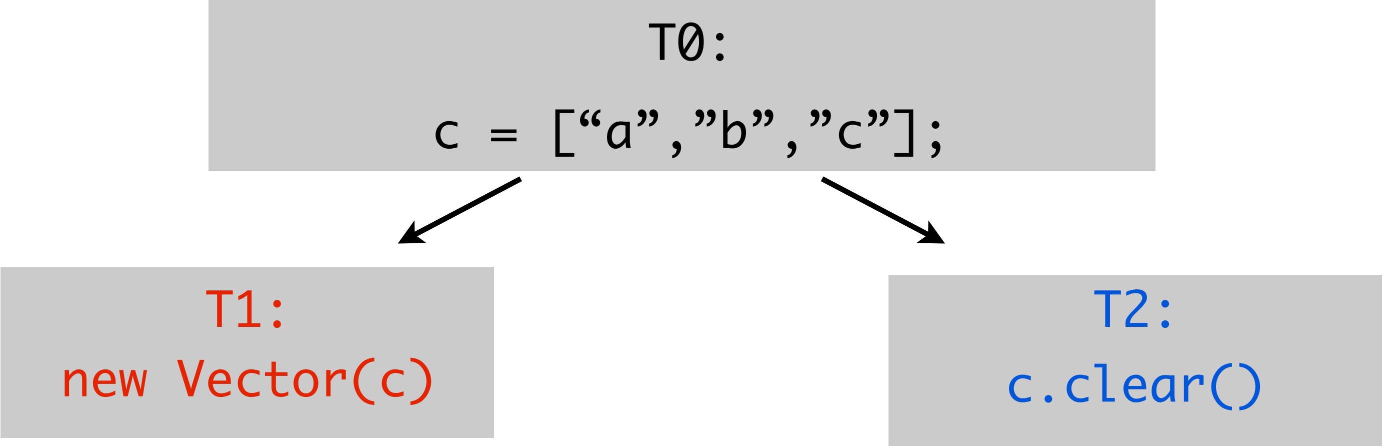
VECTOR

```
        class Vector extends ... implements ... {
          public Vector(Collection c){
            elementCount = c.size();
            elementData = new Object[(int)Math.min(
                   (elementCount*110L)/100,Integer.MAX_VALUE)];
T1 →        c.toArray(elementData);
          }

        }
```

```
elementCount = 3
elementData = [null, null, null]
c = []
```

ATOMICITY VIOLATION!

in other words, executing the following code:



```
T0:
c = [“a”,”b”,”c”];
```

```
T1:
new Vector(c)
```

```
T2:
c.clear()
```

results in T1 creating the following vector:

```
[null,null,null]
```

MANY SIMILAR ATOMICITY VIOLATIONS

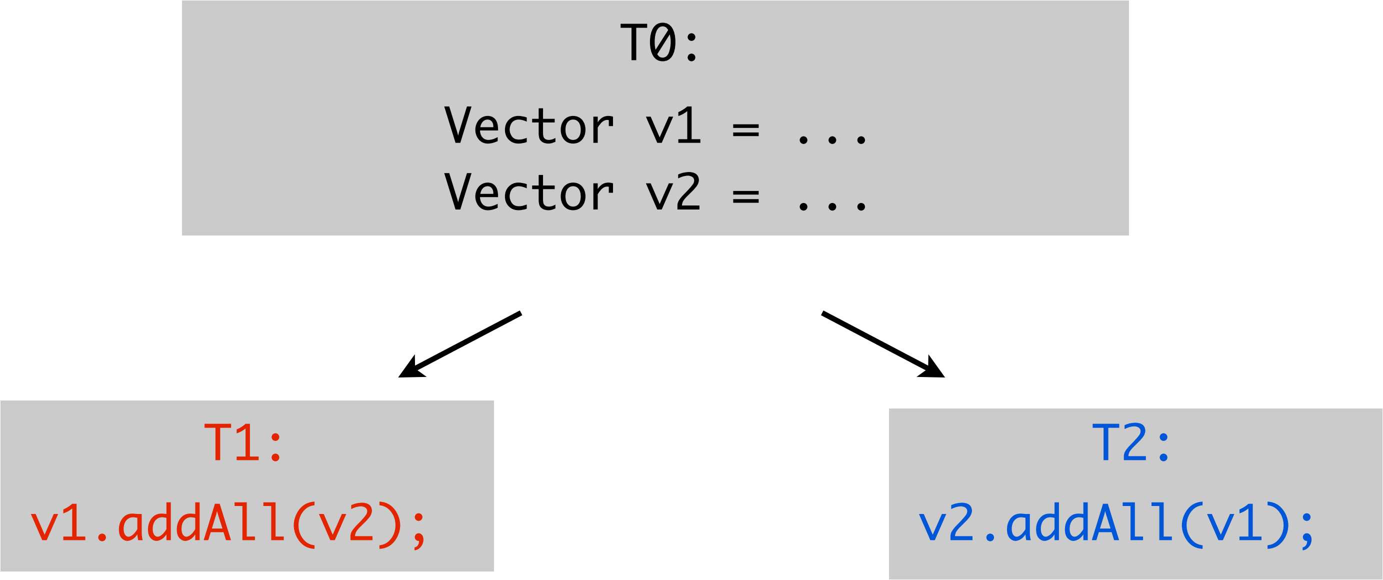
```
class Vector extends ... implements ... {
  ...

  public synchronized boolean addAll(Collection c){
    ...
  }

}
```

(NullPointerException may occur if c is modified concurrently)

PREVENTING ATOMICITY VIOLATIONS..

```
class Vector extends ... implements ... {
  ...

  public boolean addAll(Collection c){
    synchronized (this){
      synchronized (c){
        ...
      }
    }
  }

}
```

BUT...



```
T0:
Vector v1 = ...
Vector v2 = ...
```

```
T1:
v1.addAll(v2);
```

```
T2:
v2.addAll(v1);
```

```
Vector v1 = ...
Vector v2 = ...

T1 acquires lock for v1
T2 acquires lock for v2
```

```
Vector v1 = ...
Vector v2 = ...

T1 acquires lock for v1
T2 acquires lock for v2
```

- now, both threads hold one lock and are trying to acquire the other…

  ⇒ **deadlock**

PREVENTING DEADLOCK

- java.util.concurrent.ReentrantLock provides trylock() mechanism:
  - acquire first lock
  - check if second lock is available
  - if so, acquire second lock and proceed; otherwise release previously acquired lock and go back to step 1

- impose a partial ordering on locks
  - only acquire locks in accordance with specified order
  - lock ordering may be hard to define on dynamically allocated objects

- either approach leads to more convoluted and error-prone code
  - burden is on the programmer to “get it right”

HOW TO AVOID CONCURRENCY ERRORS WHEN USING THREADS

- protect shared locations with locks to prevent data races
- protect groups of shared locations to prevent atomicity violations
- acquire multiple locks in consistent order, to prevent deadlock
- must protect every access to shared data consistently

OTHER SOLUTIONS

- in Java:
  - Executors
  - Thread Pools
  - Fork/Join
  - java.util.concurrent library

- alternative approaches to concurrency in other languages
  - e.g., Scala's actors

- research topics:
  - detection of concurrency-related errors using static analysis, dynamic analysis, model checking, …