02_Locks

- Race conditions
- Synchronization with Java Language Features
- Synchronization with Locks
- Deadlocks

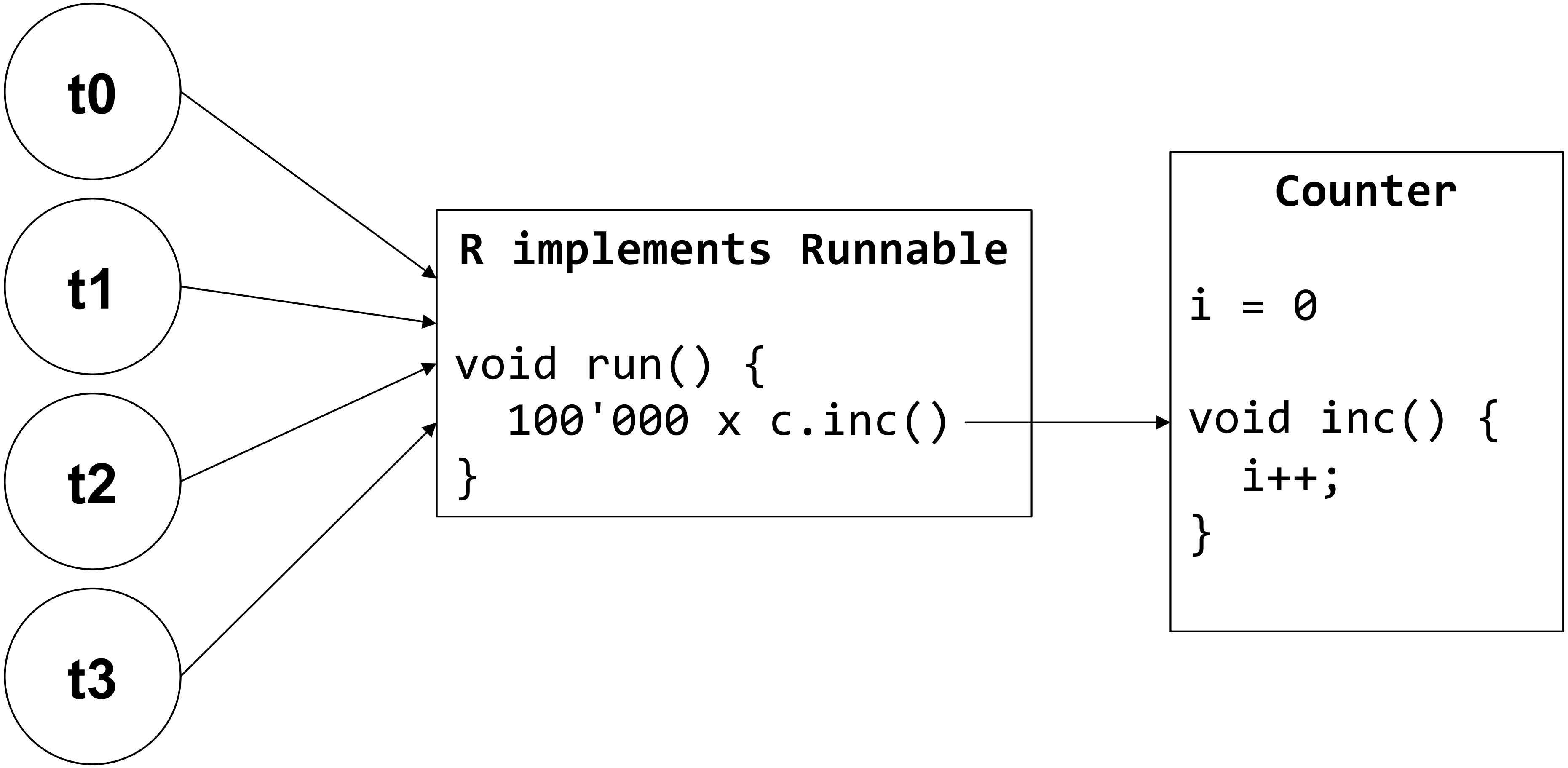
Demo Counter

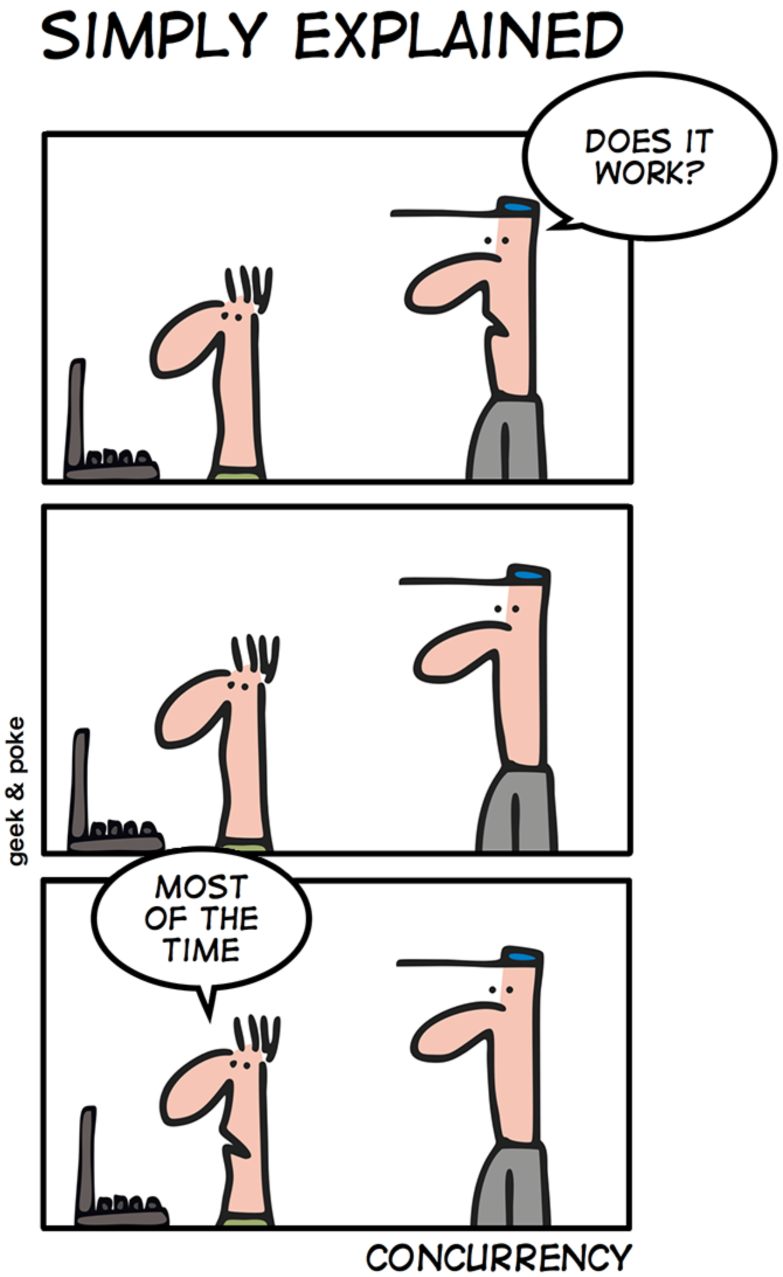
```
class Counter {
  private int i = 0;
  public void inc() { i++; }
  public int getCount() { return i; }
}

class R implements Runnable {
  private Counter c;
  public R(Counter c) { this.c = c; }

  public void run() {
    for (int i = 0; i < 100000; i++) {
      c.inc();
    }
  }
}
```

Demo Counter

```
public class CounterTest {

  public static void main(String[] args) {
    Counter c = new Counter();
    Runnable r = new R(c);
    Thread t0 = new Thread(r); Thread t1 = new Thread(r);
    Thread t2 = new Thread(r); Thread t3 = new Thread(r);
    t0.start(); t1.start(); t2.start(); t3.start();

    try {
      t0.join(); t1.join(); t2.join(); t3.join();
    } catch (InterruptedException e) {}

    System.out.println(c.getCount());
  }
}
```

Demo Counter




SIMPLY EXPLAINED
DOES IT WORK?
MOST OF THE TIME
geek & poke
CONCURRENCY

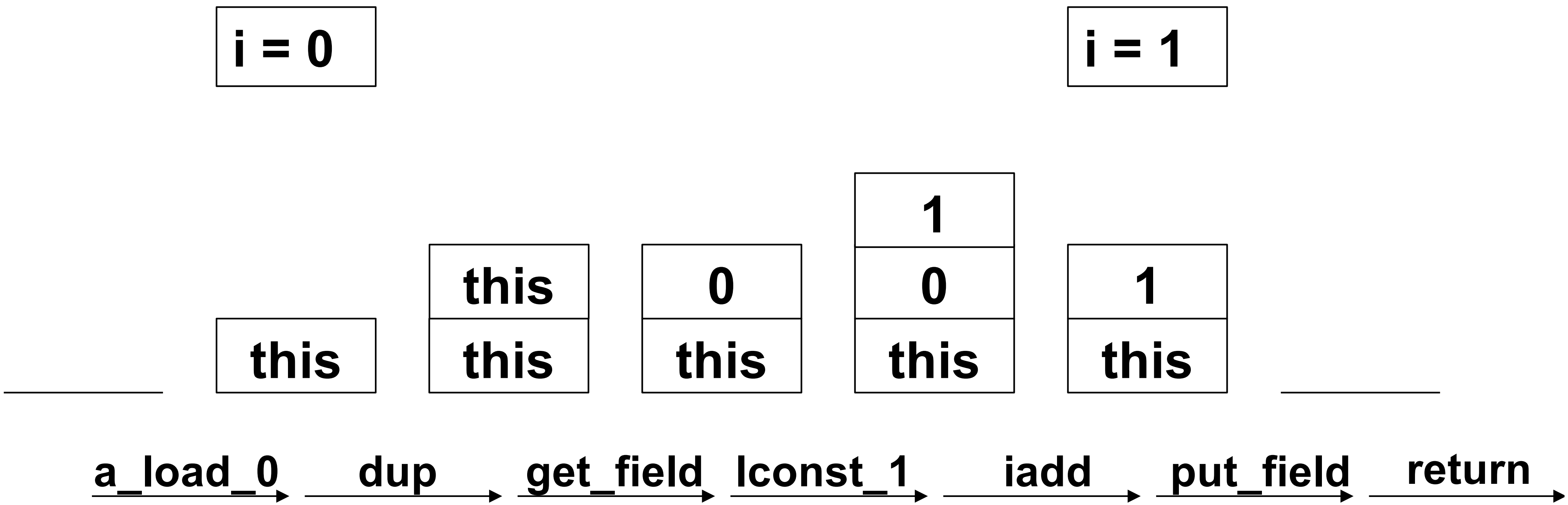
inc() Method disassembled

- **$ javap -c lecture.Counter:**

```
public class lecture.Counter {

  public void inc();
    Code:
       0: aload_0
       1: dup
       2: getfield      #2                  // Field i:I
       5: lconst_1
       6: iadd
       7: putfield      #2                  // Field i:I
      10: return
  ...
}
```

Byte Code

```
 0: aload_0
 1: dup
 2: getfield  #2 // Field i:I
 5: lconst_1
 6: iadd
 7: putfield  #2 // Field i:I
10: return
```



inc() Method disassembled

- **java -XX:+UnlockDiagnosticVMOptions**
  -XX:PrintAssemblyOptions=hsdis-printbytes
  -XX:CompileCommand=print,lecture/Counter.inc lecture.CounterTest

```
[Verified Entry Point]
  0x000000010c4ec040: mov    %eax,-0x14000(%rsp)
  0x000000010c4ec047: push   %rbp
  0x000000010c4ec048: sub    $0x30,%rsp           ;*aload_0
                                                  ; - lecture.Counter::inc@0
  0x000000010c4ec04c: mov    0xc(%rsi),%edi       ;*getfield i
                                                  ; - lecture.Counter::inc@2
  0x000000010c4ec04f: inc    %edi
  0x000000010c4ec051: mov    %edi,0xc(%rsi)       ;*putfield i
                                                  ; - lecture.Counter::inc@7
  0x000000010c4ec054: add    $0x30,%rsp
  0x000000010c4ec058: pop    %rbp
  0x000000010c4ec059: test   %eax,-0x20eef5f(%rip)        # 0x000000010a3fd100
                                                  ;   {poll_return}
  0x000000010c4ec05f: retq
```

Interleaving of Instructions

- **Scheduler is allowed to switch context between every operation**
  - Even read and write access to longs and doubles are not guaranteed to be atomic!

Thread A		i		Thread B
read i onto stack (42)	←	42		
		42	→	read i onto stack (42)
		42		top of stack + 1 (43)
top of stack + 1 (43)		42		
write top to i (43)	→	43		
		43	←	write top to i (43)

Lost Update

Interleaving Model

- **Number of all possible interleavings depending on the number of threads (n) and the number of atomic instructions (m)**

$$\#\text{interleavings} = \frac{(n * m)!}{(m!)^n}$$

- **Examples**
  - 2 threads and 3 atomic instructions $\dfrac{(2 * 3)!}{(3!)^2} = 20$
  - 4 threads and 3 atomic instructions $\dfrac{(4 * 3)!}{(3!)^4} =$ **369'600**

Fachhochschule Nordwestschweiz

Synchronization: Stack Example

- **Example: Stack**

```
public class Stack {
   private final int[] data = new int [10];
   private int top = 0;

   public void push(int x) {
      data[top] = x;
      top++;
   }
   public int pop() {
      top--;
      return data[top];
   }
}
```

  – two threads, one is pushing, the other popping objects

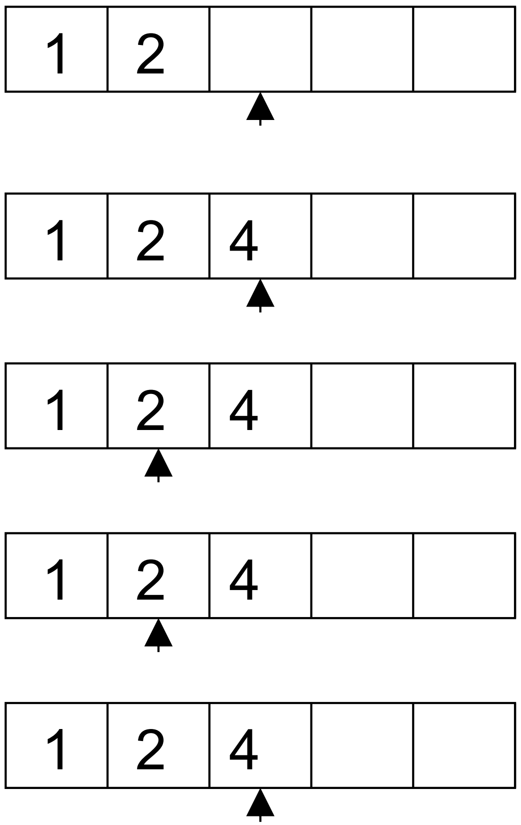
Synchronization: Stack Example

Thread A

push(4);

data[2] = 4

top++



Thread B

pop();

top--;

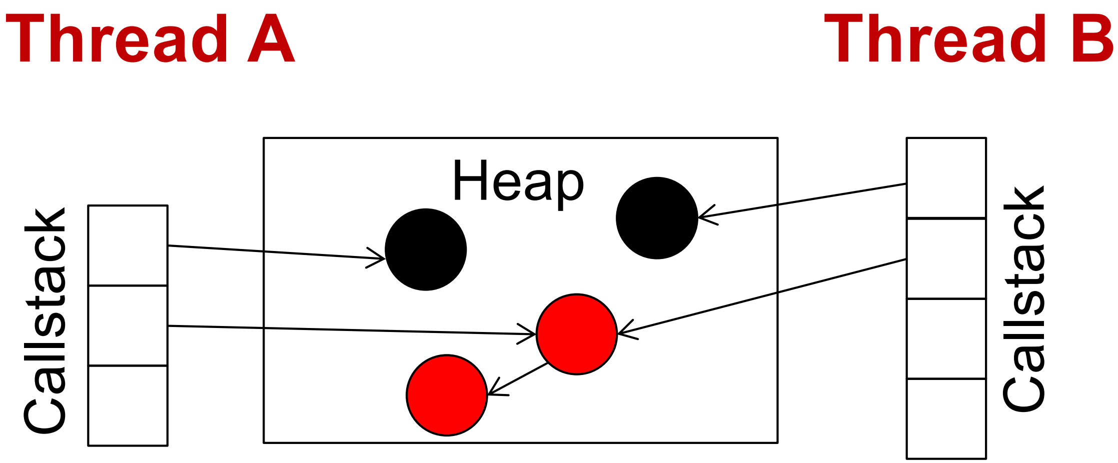
return data[1]

Race Conditions

- **Definition**
  - Two or more threads are accessing some shared data
    - At least one is modifying
  - The final result depends on the timing of how the threads are scheduled
- **Problem**
  - Thread scheduling on the JVM is nondeterministic
  - Unpredictable results and subtle program bugs
- **Solution**
  - Use proper synchronization

Synchronization

- **Managing access to shared, mutable state**
  - State: Data stored on the heap in instance- and static fields (transitively)
  - Mutable: Variable could change its value during its lifetime
  - Shared: Variable could be accessed by multiple threads



Example

```java
public class NumberRange {
  // INVARIANT: lower <= upper
  private int lower, upper = 0;

  public void setLower(int l) {
    if (l > upper) throw new IllegalArgumentException();
    lower = l;
  }

  public void setUpper(int u) {
    if (u < lower) throw new IllegalArgumentException();
    upper = u;
  }

  public boolean contains(int x) {
    return lower <= x && x <= upper;
  }
}
```

Race Condition Example

```
// INVARIANT: lower <= upper
```

Initial State: `lower = 5` | `upper = 10`

Thread 1

```
setLower(7);
```

```
void setLower(int l) {
  if (l > upper) throw new IAE();
  lower = l;
}
```

Thread 2

```
setUpper(6);
```

```
void setUpper(int u) {
  if (u < lower) throw new IAE();
  upper = u;
}
```

Post State: `lower = 7` | `upper = 6`

Invariant broken!

Object State and Invariants

- **Invariants = Properties about an objects state that *always* hold**
  - The property holds when the object is created
  - Every public method preserves the property
    - Temporary breaking between private method calls is possible
- **Preserving invariants is harder in a concurrent setting**
  - Threads may be switched in the middle of a method

  => Need some mechanism to prevent other threads from accessing an object while we're in the middle of modifying it!

> Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronization. [JCIP 2]

02_Locks

- Race conditions
- Synchronization with Java Language Features
- Synchronization with Locks
- Deadlocks

Locking in General

- **Mechanism for enforcing mutual exclusion**
  - restricting access to one thread at a time
  - guarding a critical section from concurrent execution
- **Structure:**

```
lock.lock();
  Critical Section
lock.unlock();
```

- **lock.lock()**
  - lets the *first* thread pass
  - blocks all following threads until the first thread calls lock.unlock()
- **lock.unlock()**
  - releases the lock such that the next thread can pass

Fixing the Counter

Thread A	i	Thread B
Try to lock, get lock	42	
read i onto stack (42) ←	42	
	42	Try to lock, cannot, must block
top of stack + 1 (43)	42	(blocked)
write top to i (43) →	43	(blocked)
Release lock	43	(blocked)
	43	Get lock
	43 →	read i onto stack (43)
	43	top of stack + 1 (44)
	44 ←	write top to i (44)
	44	Release lock

Locking in Java

- *Every* object contains a built-in (intrinsic) lock
- The synchronized keyword is the built-in locking mechanism for enforcing atomicity. It consists of two parts
  - reference to an object that will serve as lock
  - a block of code to be guarded by that lock

```
synchronized (lock) {
   Critical Section
}
```

=

```
lock.lock();
   Critical Section
lock.unlock();
```

- Lock is acquired when synchronized section is entered
  - if lock is not available, thread enters a waiting queue
- Lock is released when synchronized section is exited
  - also in case of exception

Fixing the Stack

- **Problem: Compound actions (data access and index adjustment) were not atomic (and thus possibly interleaved)**
- **Solution: Guard compound actions with a lock**

```
public void push(int x) {
   synchronized(this){
      data[top] = x;
      top++;
   }
}
```

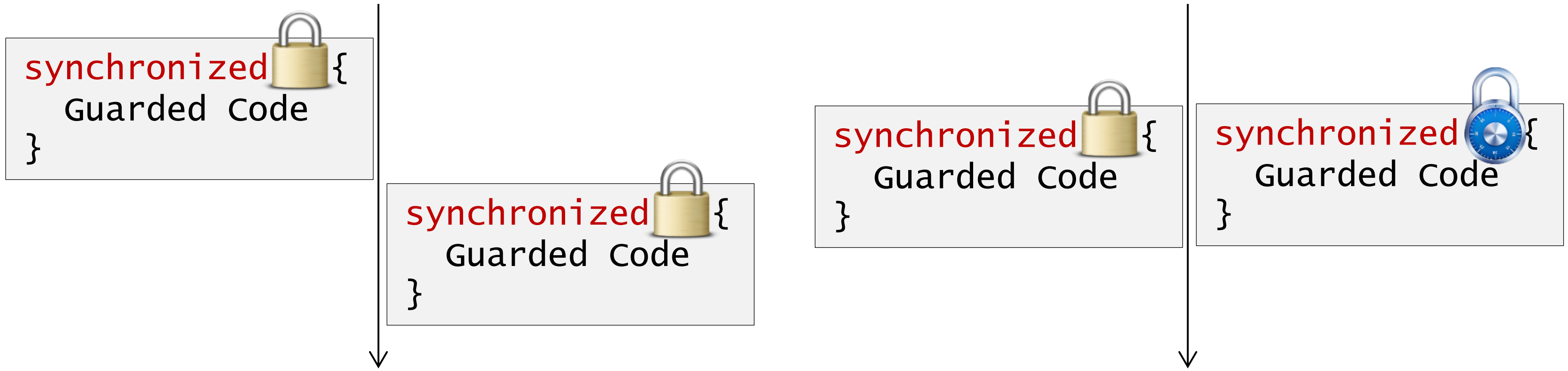
```
public int pop() {
   synchronized(this){
      top--;
      return data[top];
   }
}
```

> To preserve state consistency, update related state variables in a single atomic operation. [JCIP 2.3]

Atomicity

- **Synchronized blocks guarded by the same lock**
  - execute atomically with respect to one another
- **Synchronized blocks guarded by different locks**
  - execute non atomically and may be scheduled interleaved

Same Lock **Different Locks**



Synchronized Short Forms

- **instance methods**
  - often a method is synchronized on “this”:

```
public void push(int x) { synchronized(this) { … } }
```

  - short form

```
public synchronized void push(int x) { … }
```

- **class methods (static)**
  - A lock is also associated with each class,
    this lock is different from the locks of the instances

```
public static synchronized void foo() { … }
```

similar to

```
class C {
   public static void foo() { synchronized(C.class) { … } }
}
```

Synchronization and Data Protection

- **Data Protection**
  - Synchronized lock flag does not protect data but synchronizes threads
  - Data can still be manipulated by direct access
    => declare data as private to prevent uncontrolled access
  - Data can still be accessed by unsynchronized threads
    => synchronize all methods which can access critical data

For each mutable state variable that may be accessed by more than one thread, all access to that variable must be performed with the *same* lock held. In this case, we say that the variable is *guarded by* that lock. [JCIP 2.3]

For every invariant that involves more than one variable, *all* the variables involved in that invariant must be guarded by the *same* lock.

Synchronization in the JVM

- **Synchronized methods**

```
public synchronized int getAndIncrement(){
   return counter++;
}
```

  - A flag is set in the method table for this method

- **Synchronized blocks**

```
public int getAndIncrement(){
   synchronized(this){
      return counter++;
   }
}
```

  - Byte code instructions are added to the generated code

Synchronization in Bytecode

- **Java Code**

```
public class Test {
   public int x;
   public void foo() {
      synchronized(this) {
         x++;
      }
   }
}
```

- **Byte Code**
  - Monitorenter / monitorexit

```
public void foo() {
   monitorenter(this);
   try {
      ...
   } finally {
      monitorexit(this);
   }
}
```

```
public void foo();
  Code:
   0:    aload_0
   1:    dup
   2:    astore_1
   3:    monitorenter
   4:    aload_0
   5:    dup
   6:    getfield     #2; //Field x:I
   9:    iconst_1
   10:   iadd
   11:   putfield     #2; //Field x:I
   14:   aload_1
   15:   monitorexit
   16:   goto     24
   19:   astore_2
   20:   aload_1
   21:   monitorexit
   22:   aload_2
   23:   athrow
   24:   return
  Exception table:
   from    to  target type
     4    16    19    any
    19    22    19    any
```

Reentrancy of Synchronized

- **Reentrancy**
  - A synchronization lock can be acquired multiple times by the same thread

```
synchronized(x) {
   synchronized(x) { /* no deadlock */ }
}
```

```
synchronized f() { g(); }
synchronized g() { /* no deadlock */ }
```

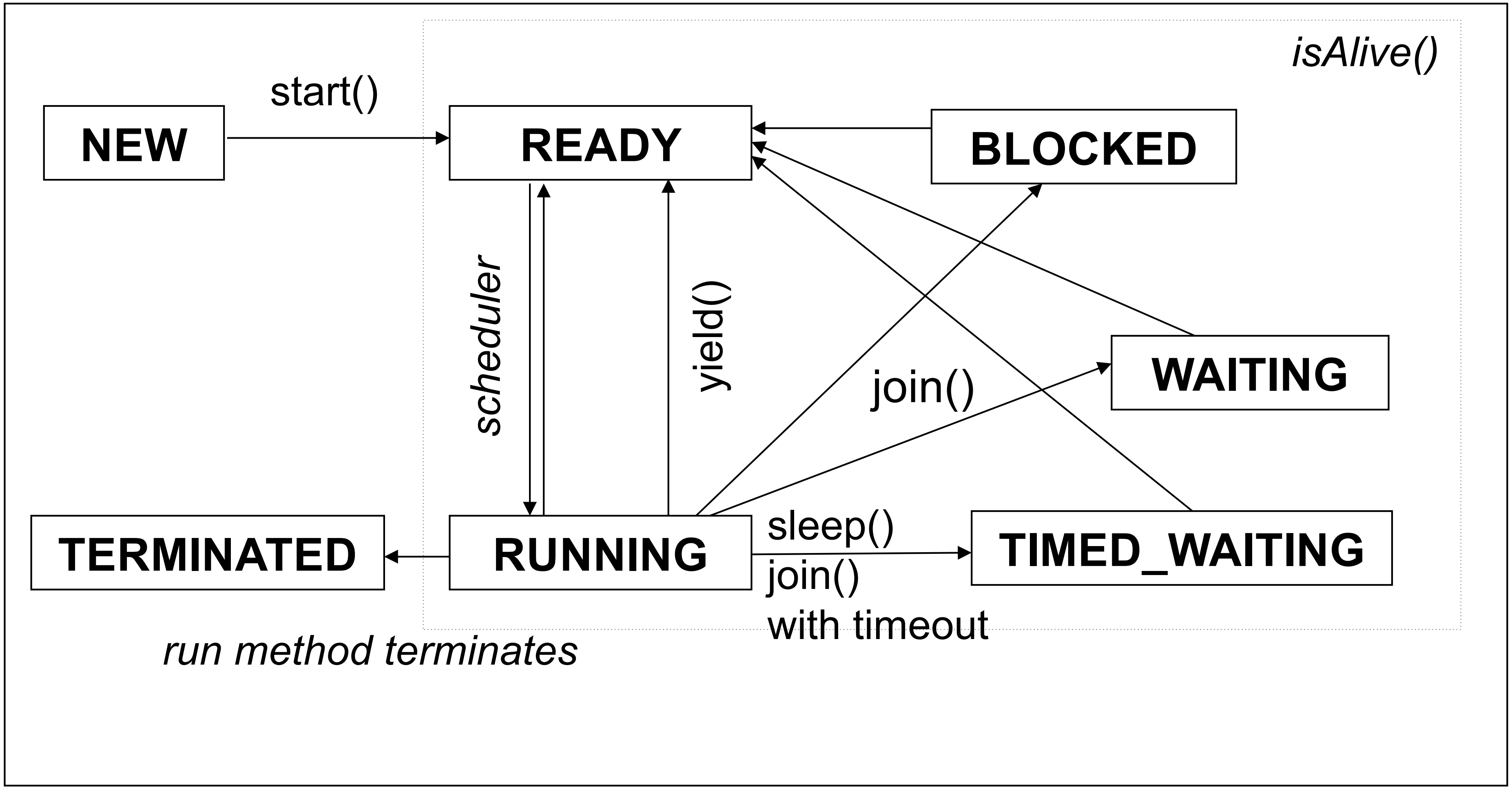
  - JVM maintains counter for each object
    - Counts number of times the object has been locked
    - Unlocked object has count 0
    - Lock (`monitorenter`): count is incremented,
      0->1: lock-id is set to current thread
    - Unlock (`monitorexit`): count is decremented;
      when it reaches 0, lock is released and made available to other threads

Synchronization and Performance

- **Performance**
  - Synchronization is not free
    - Additional code
    - Memory Barriers (=> JMM)
    - Fewer compiler / interpreter optimizations
- **Advise**
  - Compare performance gain due to multi-threading with the additional synchronization overhead
  - How can you avoid bottlenecks?
    Shared mutable state contradicts independent parallel workflows

> Avoid holding locks during lengthy computations or operations at risk of not completing quickly such as network or console I/O. [JCIP 2.5]

Thread State



Example: Copy Machine

```
class CopyMachine {
   public synchronized void makeCopies(Document d, int n) {
      // only one thread at a time !!
      Original org = scanOriginal(d);
      for(int i=0; i<n; i++) {
         Paper p = getPaper();
         while(p == null) { signalOutOfPaper(); p = getPaper(); }
         copy(org, p);
      }
   }

   public synchronized void loadPaper(Paper[] p) {
      putPaper(p);
   }
}
```

– What happens if you want to reload paper during a long print job or if you run out of paper?

Example: Copy Machine

```
class CopyMachine {
   private Object paperLock = new Object();
   public synchronized void makeCopies(Document d, int n) {
      Original org = scanOriginal(d);
      for(int i=0; i<n; i++) { Paper p = null;
         while(p == null) {
            synchronized(paperLock) { p = getPaper(); }
            if(p == null) signalOutOfPaper();
         }
         copy(org, p);
      }
   }
   public void loadPaper (Paper[] p) {
      synchronized(paperLock) {
         putPaper(p)
      }
   }
}
```

One lock at an object level may be too coarse

=> use dedicated objects as simple locks

Locking: Design Considerations

- **Which lock object should be used**
  - synchronized(this)
    - exposes possible implementation details
    - makes code vulnerable to lock attacks
  - synchronized(lock)
    - lock object may be declared private
    - more explicit
    - often preferable
- **Make the locking strategy explicit**
  - Document for maintainers which state is guarded by which lock

> Every shared, mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is. [JCIP 2.4]

Lock Attack

```
final List<Integer> l = Collections.synchronizedList(
                                               new LinkedList<Integer>());
new Thread() {
    public void run() {
        while(true) {
            l.add(1); System.out.println("Insert");
            try { Thread.sleep(1000); } catch (Exception e) {}
        }
    }
}.start();

Thread.sleep(5000);
new Thread() {
    public void run() {
        synchronized (l) {
            System.out.println("No more progress!");
            try { Thread.sleep(Long.MAX_VALUE); } catc
        }
    }
}.start();
```

Output:
Insert
Insert
Insert
Insert
Insert
No more progress!

What is the most frequent concurrency issue you've encountered in Java?

113

My #1 most painful concurrency problem ever occurred when two different open source libraries did something like this:

```
private static final String LOCK = "LOCK";  // use matching strings
                                            // in two different libraries

public doSomestuff() {
   synchronized(LOCK) {
       this.work();
   }
}
```

At first glance, this looks like a pretty trivial synchronization example. However; because Strings are interned in Java, the literal string "LOCK" turns out to be the same instance of java.lang.String (even though they are declared completely disparately from each other.) The result is obviously bad.

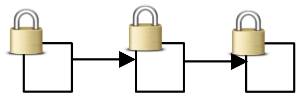
share | edit | flag   answered Jan 20 '09 at 22:44   community wiki
Jared

02_Locks

- Race conditions
- Synchronization with Java Language Features
- Synchronization with Locks
- Deadlocks

java.util.concurrent.locks.Lock

- **Provides more flexible locking operations than *synchronized***
  - Fairness
  - Non-block locking structures
    - Hand-over-hand locking
    - Lock may be acquired and released in different scopes
  - Thread may acquire a lock timed
  - Thread may test if lock is available and acquire it atomically



- **Use `java.util.concurrent.locks.Lock` when `synchronized` proves too limited**

java.util.concurrent.locks.Lock

- **Interface**

```
public interface Lock {
   void lock();
   void unlock();

   boolean tryLock();
   boolean tryLock(long timeout, TimeUnit unit)
                                  throws InterruptedException;

   ...
}
```

java.util.concurrent.locks.Lock

- **Usage pattern**

```
Lock lock = ...;
...
lock.lock();
try {
   // access resources protected by this lock
}
finally {
   lock.unlock();
}
```

  - Additional responsibility for the programmer
  - FindBugs has a "unreleased lock" detector

java.util.concurrent.locks.ReentrantLock

- **Class ReentrantLock implements interface Lock**
  - *lock* returns immediately if the current thread already holds the lock (only a lock counter is incremented)
  - Additional methods
    - getOwner — returns thread that currently holds the lock
    - isHeldByCurrentThread — queries if this lock is held by the current thread
    - getHoldCount — returns lock counter hold by current thread
    - getQueueLength — number of waiting threads (estimate)
  - Lock has to be released by the same thread which acquired the lock

```
public static void main(String[] args) {
   final Lock l = new ReentrantLock();
   l.lock();
   new Thread(){ public void run(){ l.unlock(); }}.start();
}
```

java.util.concurrent.locks.ReentrantLock

- **Fairness: ReentrantLock offers two options**
  - Fair
    - Threads acquire lock in the order it was requested, i.e. unlock provides access to the longest waiting thread (FIFO)
    - A newly requesting thread is queued if the lock is held by another thread or if threads are queued waiting for the lock
  - Unfair [Default]
    - Barging is allowed, thus may be more efficient
    - A newly requesting thread is queued only if the lock is currently held

    - Method tryLock always barges, even for fair locks;
      if you want to honor the fairness then use tryLock(0, TimeUnit.SECONDS)

02_Locks

- Race conditions
- Synchronization with Java Language Features
- Synchronization with Locks
- Deadlocks

Deadlock

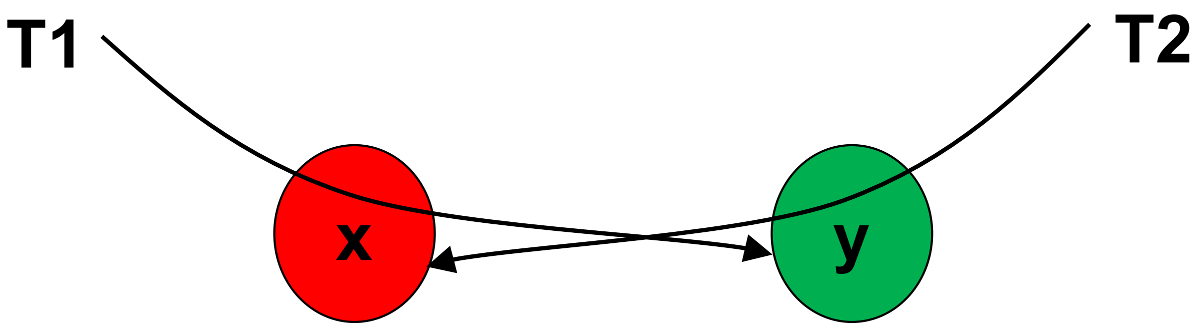
- **Example**

Thread 1

```
synchronized(x) {
    synchronized(y) {
        // access x and y
    }
}
```

Thread 2

```
synchronized(y) {
    synchronized(x) {
        // access x and y
    }
}
```



Deadlock

- **Necessary conditions for a deadlock**
  - Mutual Exclusion
    - Access to resources is exclusive
  - Hold and Wait
    - Threads request additional resources while holding resources
  - No Preemption
    - Resources are released exclusively by threads
  - Circular Wait
    - Two or more processes form a circular chain where each thread waits for a resource that the next thread in the chain holds

[Coffmann, System Deadlocks, Computing Surveys V3, p67-78, 1971]

Deadlock Prevention

- **Some conditions may be eliminated:**
  - Mutual Exclusion
    - No, is a precondition
  - Hold and Wait:
    - Acquire all needed resources at once => not possible
    - Apply a tryLock and release resources if not all can be acquired
      => not possible for synchronized
  - No preemption
    - Lock-free algorithms and STM: Rollback is performed
  - Cyclic Dependencies
    - Define a global order on resources and acquire resources ordered

Deadlock Avoidance : Global Lock Order

T1

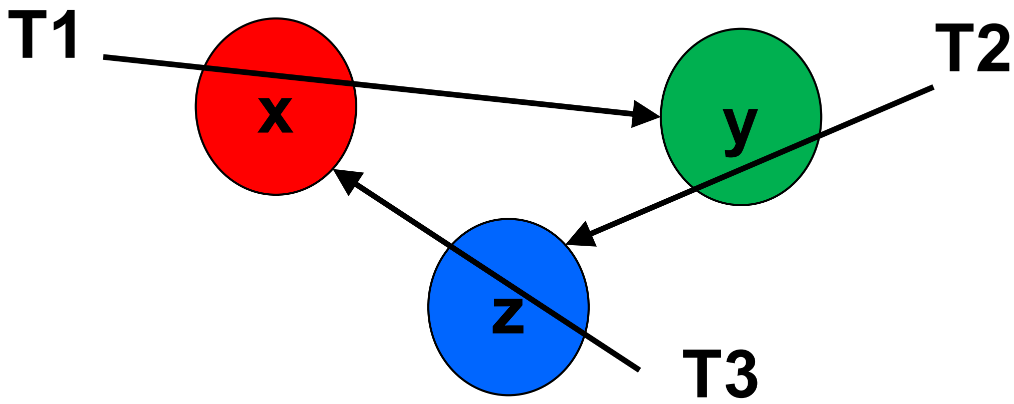
```
synchronized(x) {
   synchronized(y) {
      // access x and y
   }
}
```

T2

```
synchronized(y) {
   synchronized(z) {
      // access y and z
   }
}
```

T3

```
synchronized(z) {
   synchronized(x) {
      // access z and x
   }
}
```



Deadlock Avoidance : Global Lock Order

T1

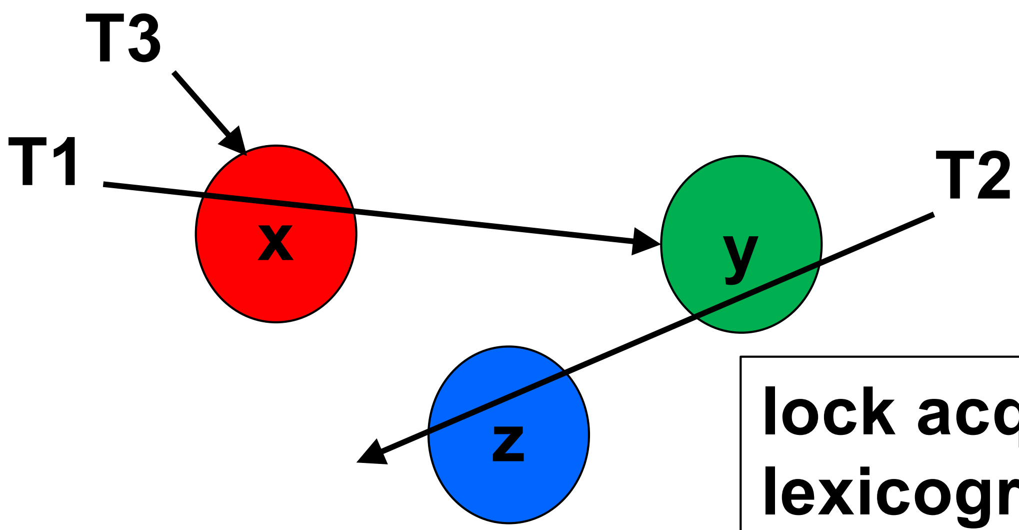
```
synchronized(x) {
  synchronized(y) {
    // access x and y
  }
}
```

T2

```
synchronized(y) {
  synchronized(z) {
    // access y and z
  }
}
```

T3

```
synchronized(x) {
  synchronized(z) {
    // access x and z
  }
}
```



lock acquisition in lexicographical order

02_Locks: Summary

Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronization. [JCIP 2]

To preserve state consistency, update related state variables in a single atomic operation. [JCIP 2.3]

For each mutable state variable that may be accessed by more than one thread, all access to that variable must be performed with the *same* lock held. In this case, we say that the variable is *guarded by* that lock. [JCIP 2.3]

Every shared, mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is. [JCIP 2.4]

Avoid holding locks during lengthy computations or operations at risk of not completing quickly such as network or console I/O. [JCIP 2.5]