Synchronization in Java

Nelson Padua-Perez

Bill Pugh

Department of Computer Science
University of Maryland, College Park

Synchronization Overview

- Unsufficient atomicity
- Data races
- Locks
- Deadlock
- Wait / Notify

Unsufficient atomicity

- Very frequently, you will want a sequence of actions to be performed atomically or indivisibly
  - not interrupted or disturbed by actions by any other thread
- x++ isn’t an atomic operation
  - it is a read followed by a write
- Can be a intermittent error
  - depends on exact interleaving

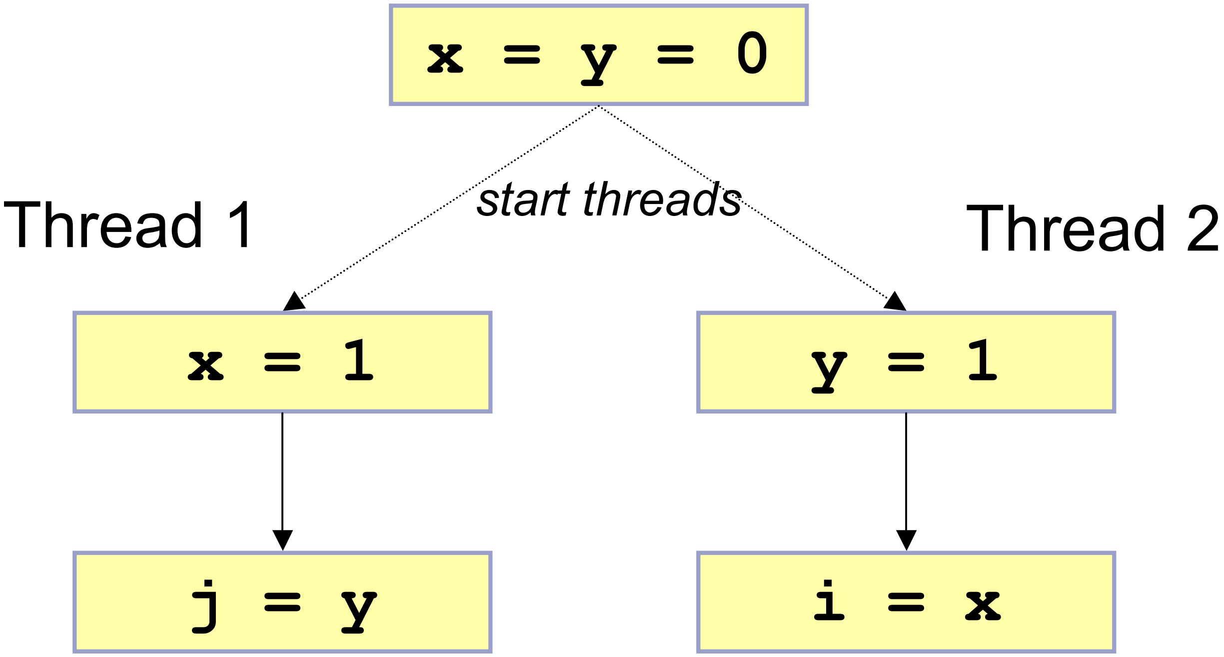
Insuffient Atomicity Example

```
public class InsuffientAtomicity implements Runnable {
    static int x = 0;
    public void run() {
        int tmp = x;
        x = tmp+1;
    }
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++)
            new Thread(new InsuffientAtomicity ()).start();
        System.out.println(x); // may not be 3
    }
}
```

Data Race

- Definition
  - Concurrent accesses to same shared variable, where at least one access is a write
    - variable isn't volatile
- Can expose all sorts of really weird stuff the compiler and processor are doing to improve performance

Quiz Time



`x = y = 0`

start threads

Thread 1

`x = 1`

`j = y`

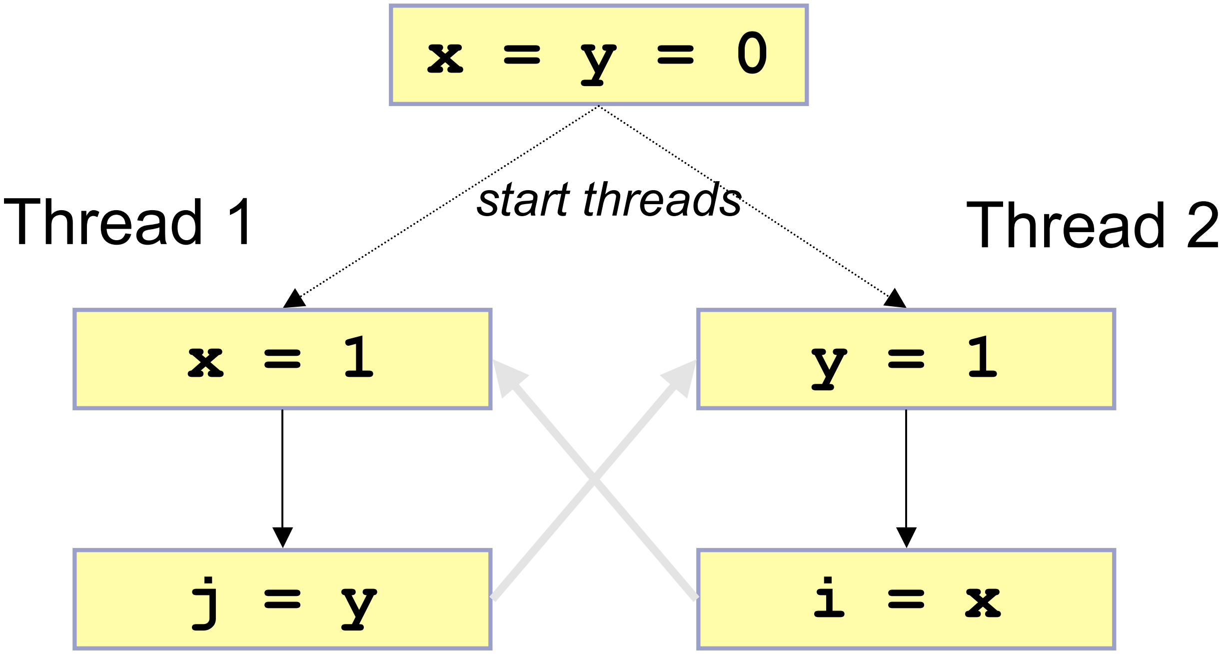
Thread 2

`y = 1`

`i = x`

Can this result in i = 0 and j = 0?

Answer: Yes!



How can i = 0 and j = 0?

How Can This Happen?

- Compiler can reorder statements
  - Or keep values in registers
- Processor can reorder them
- On multi-processor, values not synchronized to global memory
- The memory model is designed to allow aggressive optimization
  - including optimizations no one has implemented yet
- Good for performance
  - bad for your intuition about insufficiently synchronized code

Synchronization

- Uses
  - Marks when a block of code must not be interleaved with code executed by another thread
  - Marks when information can/must flow between threads
- Notes
  - Incurs a small amount of runtime overhead
    - if only used where you might need to communicate between threads, not significant
    - used everywhere, can add up

Lock

- Definition
  - Entity can be held by only one thread at a time
- Properties
  - A type of synchronization
  - Used to enforce **mutual exclusion**
  - Thread can acquire / release locks
  - Thread will wait to acquire lock (stop execution)
    - If lock held by another thread

Synchronized Objects in Java

- All Java objects provide locks
  - Apply synchronized keyword to object
  - Mutual exclusion for code in synchronization block
- Example

```
 Object x = new Object();
void foo() {
 synchronized(x) {    // acquire lock on x on entry
    ...               // hold lock on x in block
 }                    // release lock on x on exit
```

block

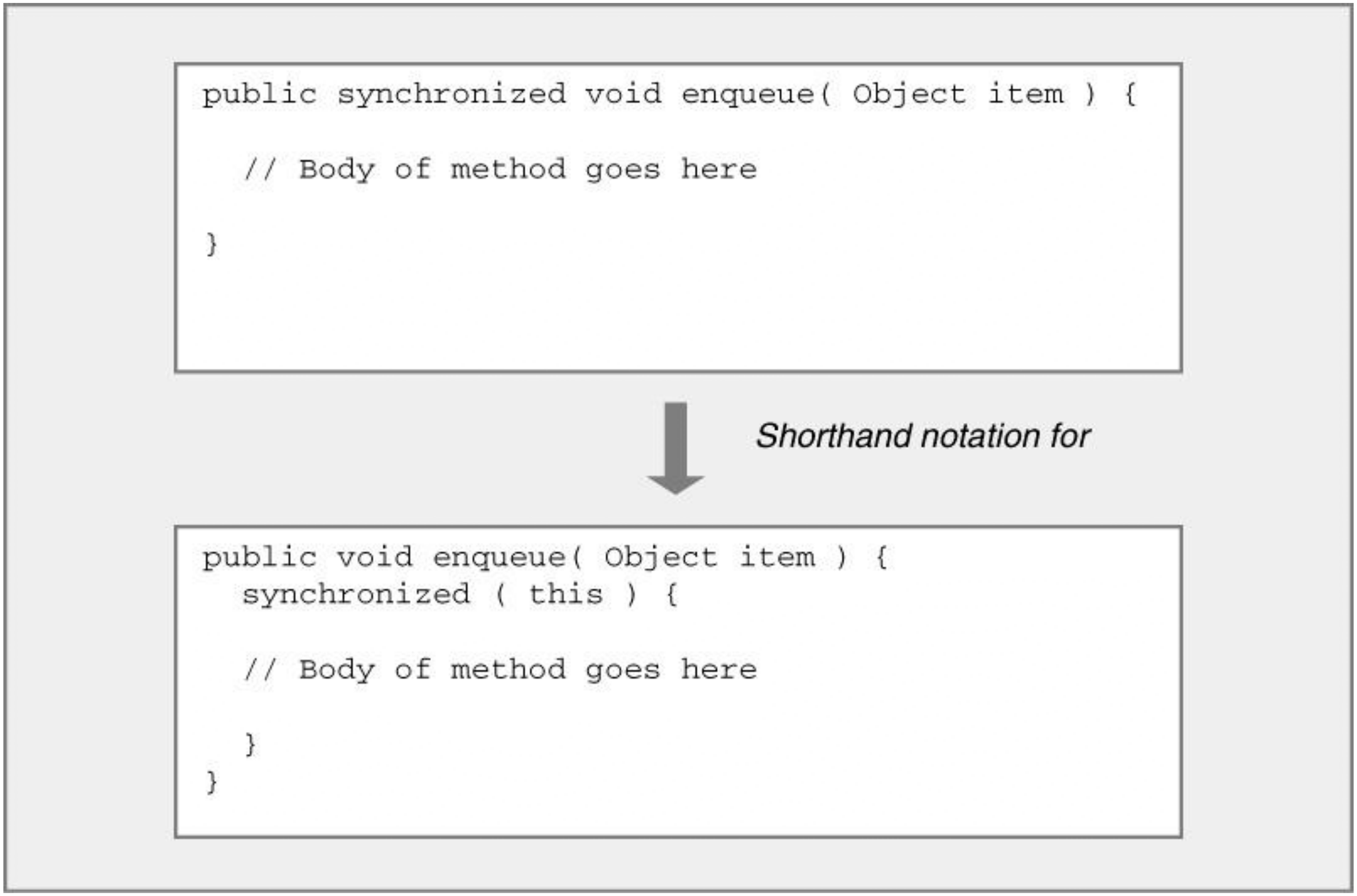
Synchronized Methods In Java

- Java methods also provide locks
  - Apply synchronized keyword to method
  - Mutual exclusion for entire body of method
  - Synchronizes on object invoking method
- Example

block

```
synchronized void foo() { …code… }
                            // shorthand notation for
void foo() {
   synchronized (this) { …code… }
}
```

Synchronized Methods In Java



```
public synchronized void enqueue( Object item ) {

   // Body of method goes here

}
```

Shorthand notation for

```
public void enqueue( Object item ) {
   synchronized ( this ) {

   // Body of method goes here

   }
}
```

Locks in Java

- Properties
  - No other thread can get lock on x while in block
  - Does not protect fields of x
    - except by convention
    - other threads can access/update fields
    - but can’t obtain lock on x
  - By convention, lock x to obtain exclusive access to x
  - Locked block of code ⇒ critical section
- Lock is released when block terminates
  - No matter how the block terminates:
    - End of block reached
    - Exit block due to return, continue, break
    - Exception thrown

Using synchronization

```
public class UseSynchronization implements Runnable {
    static int x = 0;
   static Object lock = new Object();
    public void run() {
       synchronized(lock) {
          int tmp = x;
          x = tmp+1;
          }
    }
}
```

Questions

- What would happen if the lock field were not static?
- Why don’t we just make the run method synchronized?
- Why don’t we just synchronize on x?

Not sharing same lock

```
public class NotSharingLock implements Runnable {
   static int x = 0;
   Object lock = new Object();
   public void run() {
      synchronized(lock) {
         int tmp = x;
         x = tmp+1;
         }
   }
}
```

Synchronization Issues

- Use same lock to provide mutual exclusion
- Ensure atomic transactions
- Avoiding deadlock

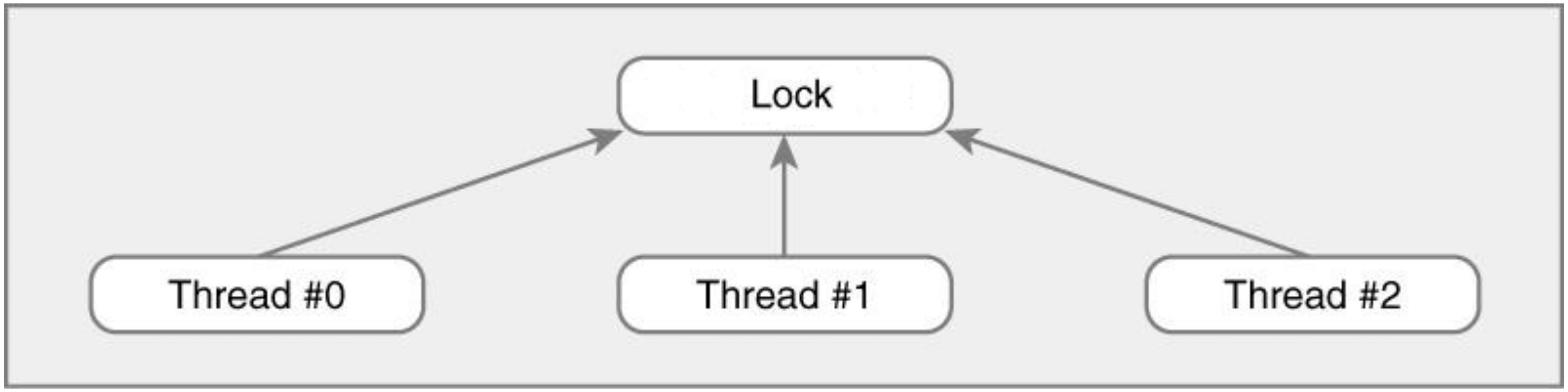
Issue 1) Using Same Lock

- Potential problem
  - Mutual exclusion depends on threads acquiring same lock
  - No synchronization if threads have different locks
- Example

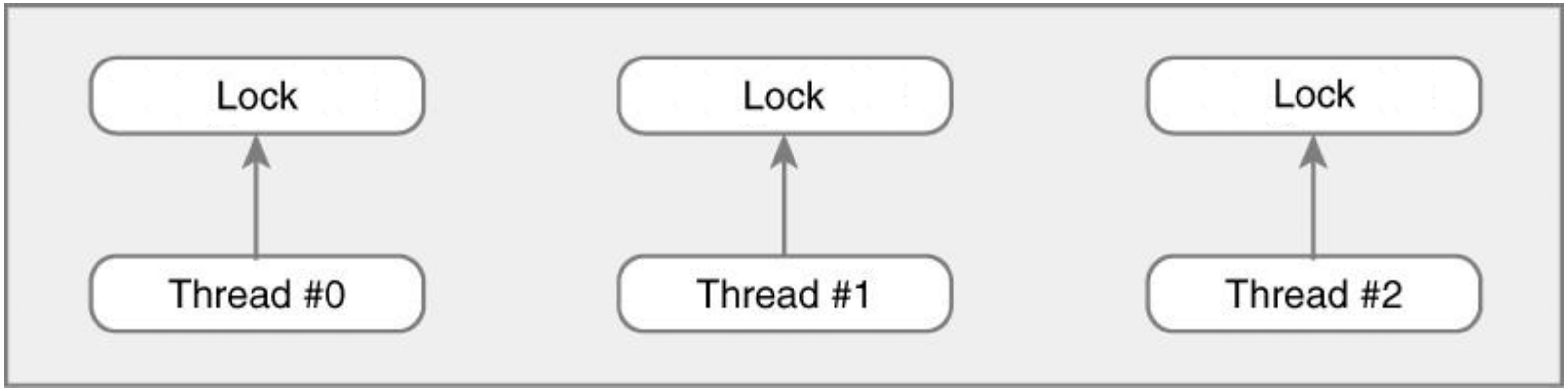
```
void run() {
    Object o = new Object(); // different o per thread
    synchronized(o) {
        … // potential data race
    }
}
```

Locks in Java

- **Single lock for all threads (mutual exclusion)**



- **Separate locks for each thread (no synchronization)**



Issue 2) Atomic Transactions

- Potential problem
  - Sequence of actions must be performed as single atomic transaction to avoid data race
  - Ensure lock is held for duration of transaction
- Example

```
synchronized(lock) {
    int tmp = x;      // both statements must
                      // be executed together
    x = tmp;          // by single thread
}
```

Using synchronization

```
public class InsuffientAtomicity implements
  Runnable {
   static int x = 0;
  static Object lock = new Object();
   public void run() {
      int tmp;
      synchronized(lock) {
       tmp = x;
     };
     synchronized(lock) {
       x = tmp+1;
       }
   }
```

Issue 3) Avoiding Deadlock

- In general, want to be careful about performing any operations that might take a long time while holding a lock
- What could take a really long time?
  - getting another lock

- Particularly if you get deadlock

Deadlock Example 1

```
Thread1() {
    synchronized(a) {
        synchronized(b) {
            …
        }
    }
}
```

```
Thread2() {
    synchronized(b) {
        synchronized(a) {
            …
        }
               }
}
```

```
// Thread1 holds lock for a, waits for b
// Thread2 holds lock for b, waits for a
```

Deadlock Example 2

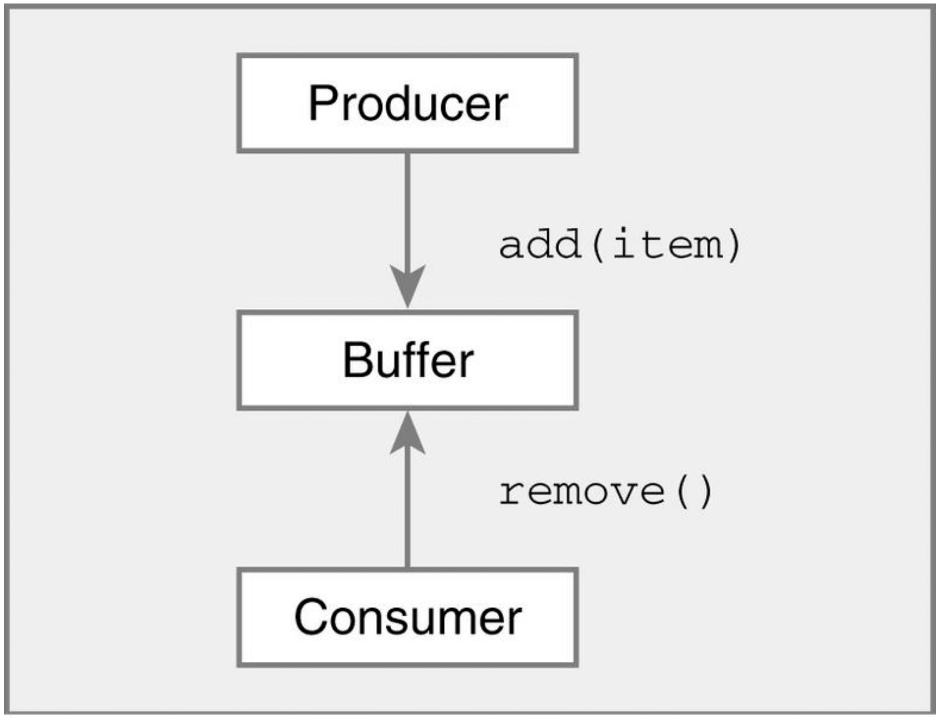
```
void moveMoney(Account a, Account b, int amount) {
    synchronized(a) {
        synchronized(b) {
            a.debit(amount);
            b.credit(amount);
        }
    }
}

Thread1() { moveMoney(a,b,10);  }
   // holds lock for a, waits for b

Thread2() { moveMoney(b,a,100);  }
   // holds lock for b, waits for a
```

Waiting for Godot

- Sometimes, you need to wait for another thread else to do something before you can do something

Abstract Data Type – Buffer

- Buffer
  - Transfers items from producers to consumers
  - Very useful in multithreaded programs
  - Synchronization needed to prevent multiple consumers removing same item



Buffer usage

- Producer thread
    - calls buffer.add(o)
    - adds o to the buffer
- Consumer thread
    - calls buffer.remove()
    - if object in buffer, removes and returns it
    - otherwise, waits until object is available to remove

Buffer Implementation

```
public class Buffer {
    private LinkedList objects = new LinkedList();
    public synchronized add( Object x ) {
        objects.add(x);
    }
    public synchronized Object remove() {
        while (objects.isEmpty()) {
            ;  // waits for more objects to be added
        }
        return objects.removeFirst();
    }
}  // if empty buffer, remove() holds lock and waits
   //   prevents add() from working ⇒ deadlock
```

Eliminating Deadlock

```
public class Buffer {
    private Object [] myObjects;
    private int numberObjects = 0;
    public synchronized add( Object x ) {
            objects.add(x);
        }
    }
    public Object remove() {
        while (true) {  // waits for more objects to be added
            synchronize(this) {
                if (!objects.isEmpty()) {
                    return objects.removeFirst(); }
            }
        }
    } // if empty buffer, remove() gives
      // up lock for a moment
```

Works barely, if at all

- Might work
- But waiting thread is going to be running a full tilt, twiddling its thumbs, doing nothing
    - burning up your battery life
    - keeping the producer from getting the CPU time it needs to quickly produce a new object

Issue 4) Using Wait & Notify

- **Potential problem**
  - Threads actively waiting consume resources

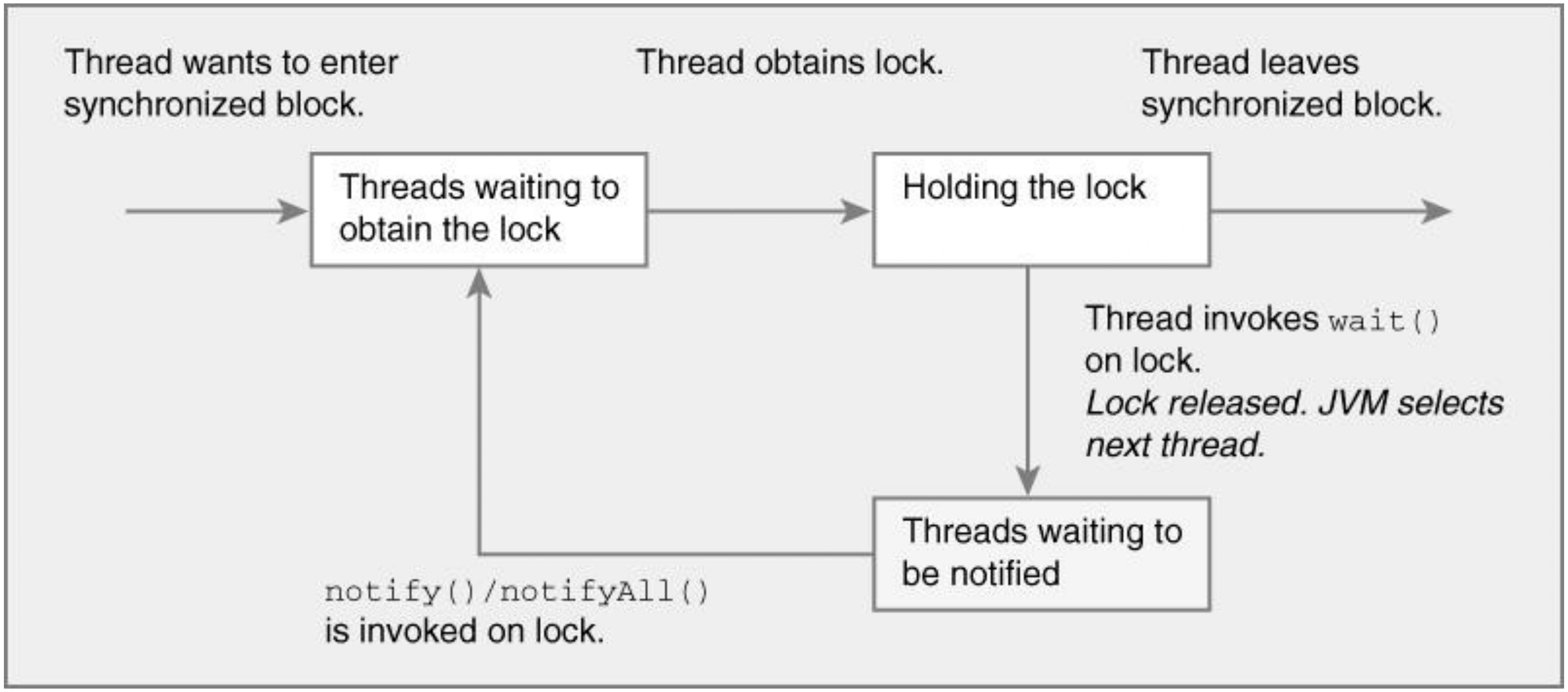
- **Solution**
  - Can wait to be notified
  - Use Thread class methods wait(), notifyAll()
    - notify() is for advanced use and tricky to get right; avoid it

Thread Class Wait & Notify Methods

- wait()
  - Invoked on object
  - must already hold lock on that object
  - gives up lock on that object
  - goes into a wait state
- notifyAll()
  - Invoked on object
  - must already hold lock on that object
  - all threads waiting on that object are woken up
    - but they all gave up their lock when they performed wait
    - will have to regain lock before then can run
    - thread performing notify holds lock at the moment

Using Wait & Notify

- **State transitions**



Thread wants to enter synchronized block.

Thread obtains lock.

Thread leaves synchronized block.

Threads waiting to obtain the lock

Holding the lock

Thread invokes `wait()` on lock.
Lock released. JVM selects next thread.

Threads waiting to be notified

`notify()/notifyAll()` is invoked on lock.

Using Wait and NotifyAll

```
public class Buffer {
   private LinkedList objects = new LinkedList();
   public synchronized add( Object x ) {
      objects.add(x);
      this.notifyAll();
   }
   public synchronized Object remove() {
      while (objects.isEmpty()) {
            this.wait();
      }
      return objects.removeFirst();
   }
}
```

Actually, that won't compile

- the wait() method is declared to throw an InterruptedException
  - a checked exception

- You rarely have situations where a wait will throw an InterruptedException
  - but the compiler forces you to deal with it

Using Wait and NotifyAll

```
public class Buffer {
   private LinkedList objects = new LinkedList();
   public synchronized add( Object x ) {
      objects.add(x);
      this.notifyAll();
   }
   public synchronized Object remove() {
      while (objects.isEmpty()) {
            try {
              this.wait();
            } catch (InterruptedException e) {}
      }
      return objects.removeFirst();
   }
}
```