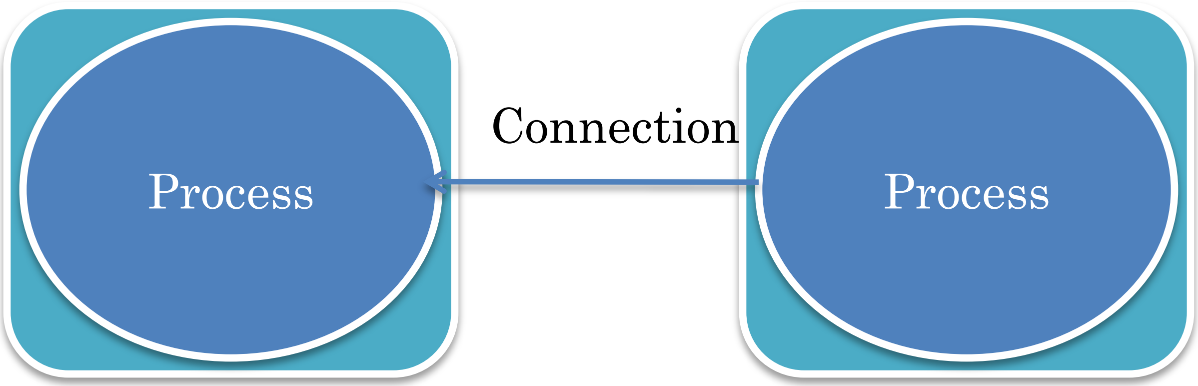
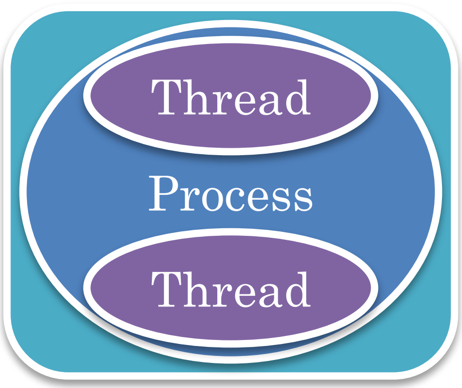
Threads and Thread Synchronization

Instructor: Prasun Dewan (FB 150, dewan@unc.edu)

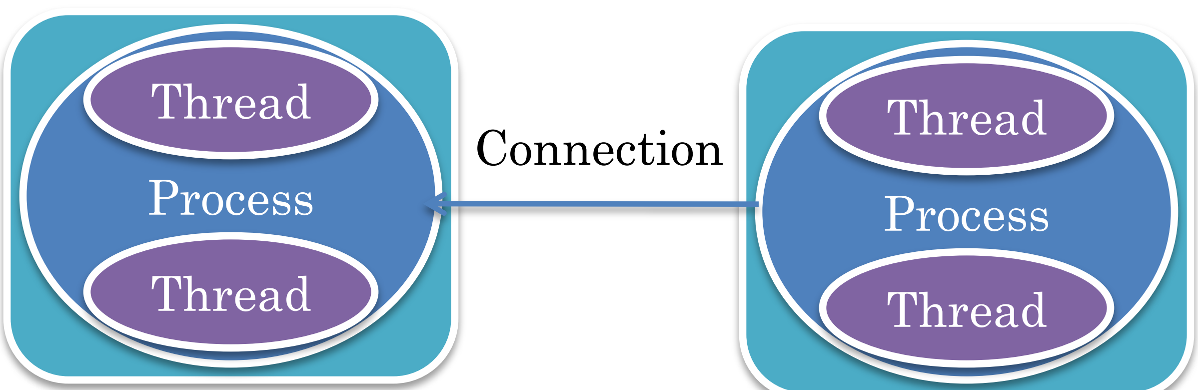
Distribution vs. Concurrency Program



Distribution, no fine-grained concurrency

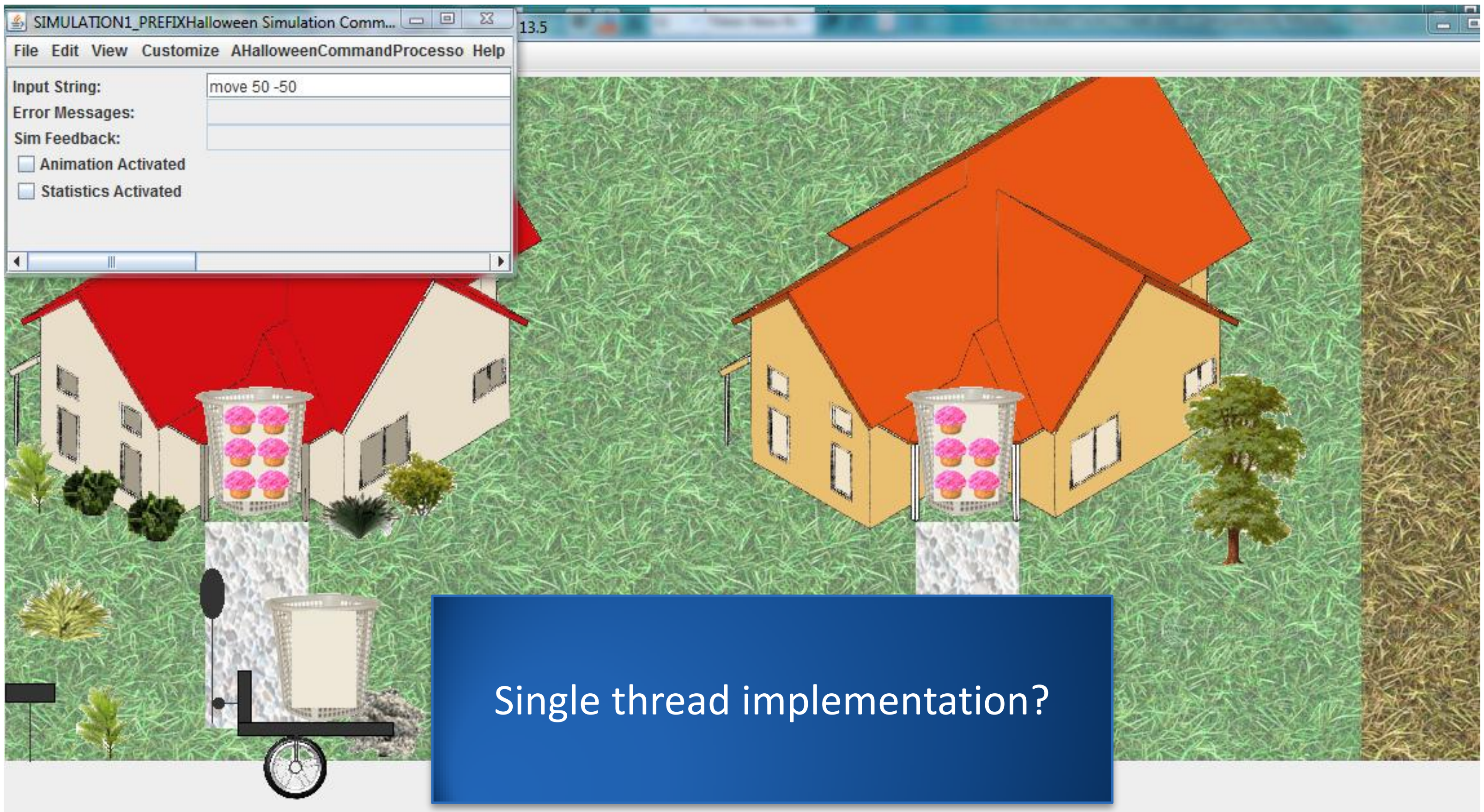


Concurrency, not distribution



Distribution and fine-grained concurrency (typical)

Demo: Halloween Simulation



Single thread implementation?

Waiting Single-Thread Solution

```
loop
   wait for local user input
   process user input
   wait for remote user input
   process remote user input
end
```

Should not impose order on user input

Not a chess game!

Polling Single-Thread Solution

```
loop
   if  (local user input received)
       process user input
   if (remote user input received
        process remote user input
    sleep (interval)
end
```

Busy waiting

Wastes computer resources

Reduces response time based on sleep time

Multiple Threads

```
loop
   wait for local user input
   process user input
end
```

```
loop
   wait for remote user input
   process remote user input
end
```

Much cleaner code

One reason threads were added to the OS Kernel

Threads and Syncyronization

Programming: Abstraction use

Systems: Abstraction design and implementation

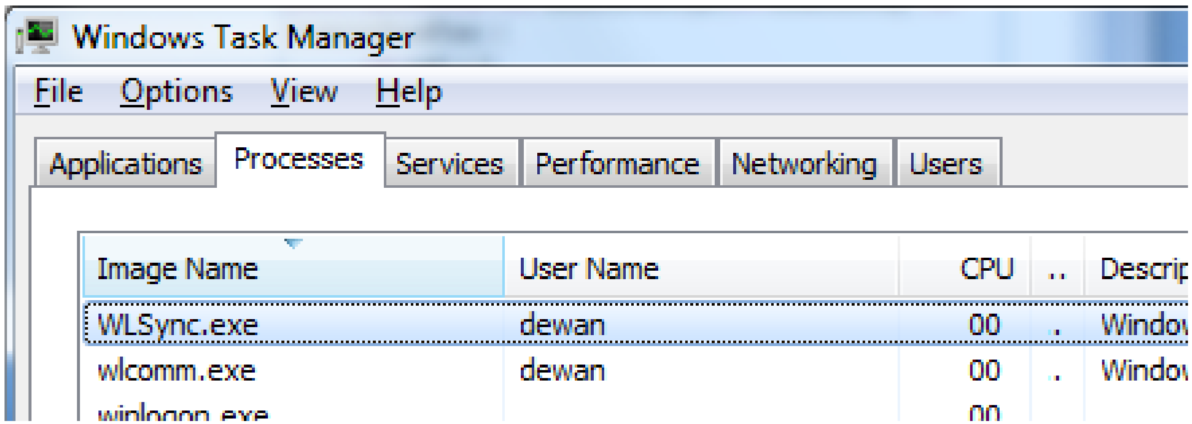
Theory: Models and algorithms

Some repetition for those who have seen threads and synchronization before

Topics

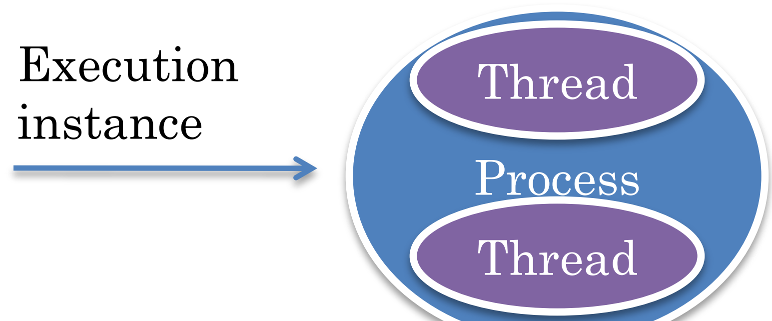
- Threads
- Command objects
- Bounded Buffer
- Generics
- Interrupts
- Critical sections
- Busy waiting
- Semaphores
- Monitors
  - Entry procedures
  - Conditions
    - Hints and Absolutes
    - Java and General Hints
- Path Expressions

PROGRAM VS. PROCESS VS. THREAD



Program

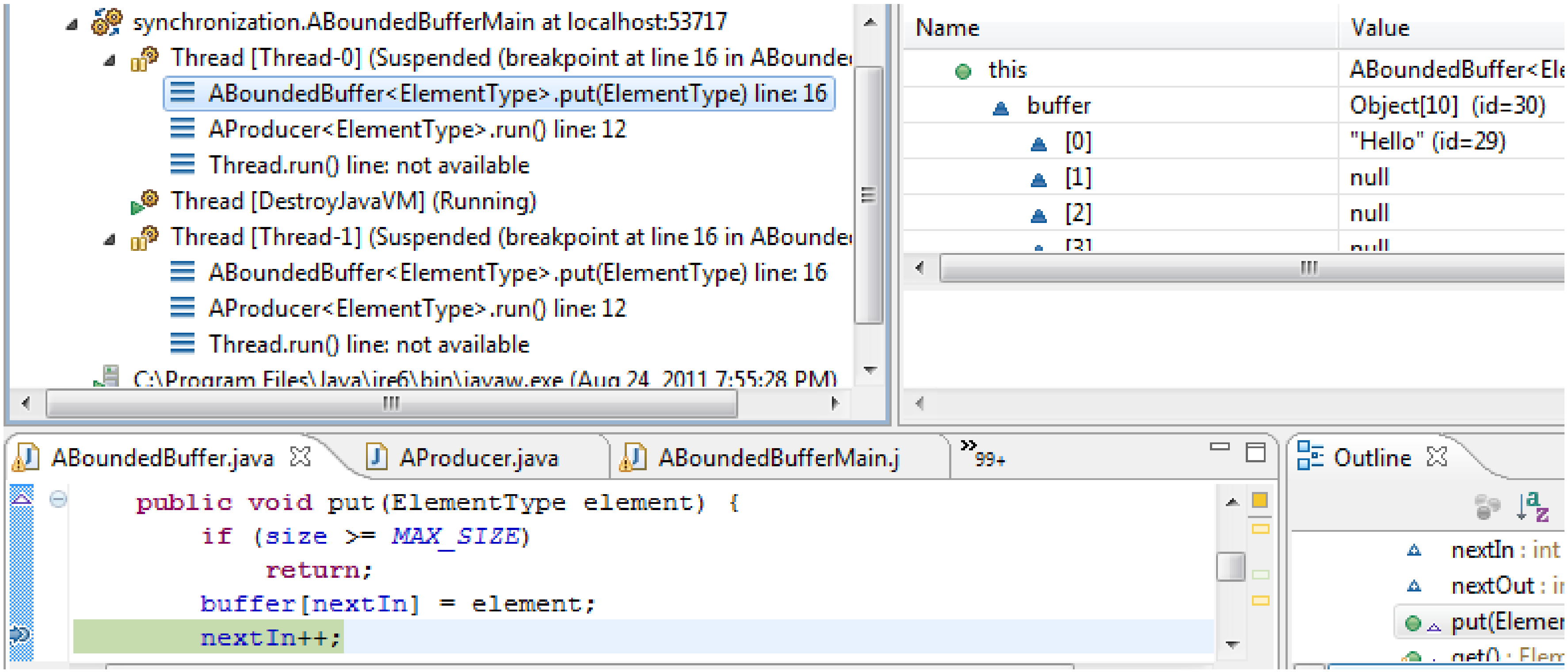
```
public class AP2PAliceTOTSimulation {
    public static String ID = "9100";
    public static String NAME = "Alice";
    public static int USER_NUMBER = 0;
    public static void main (String[] args) {
        Tracer.showInfo(true);
        AP2PTOTSessionsClientCreator.createP2PDelayed.
    }
}
```



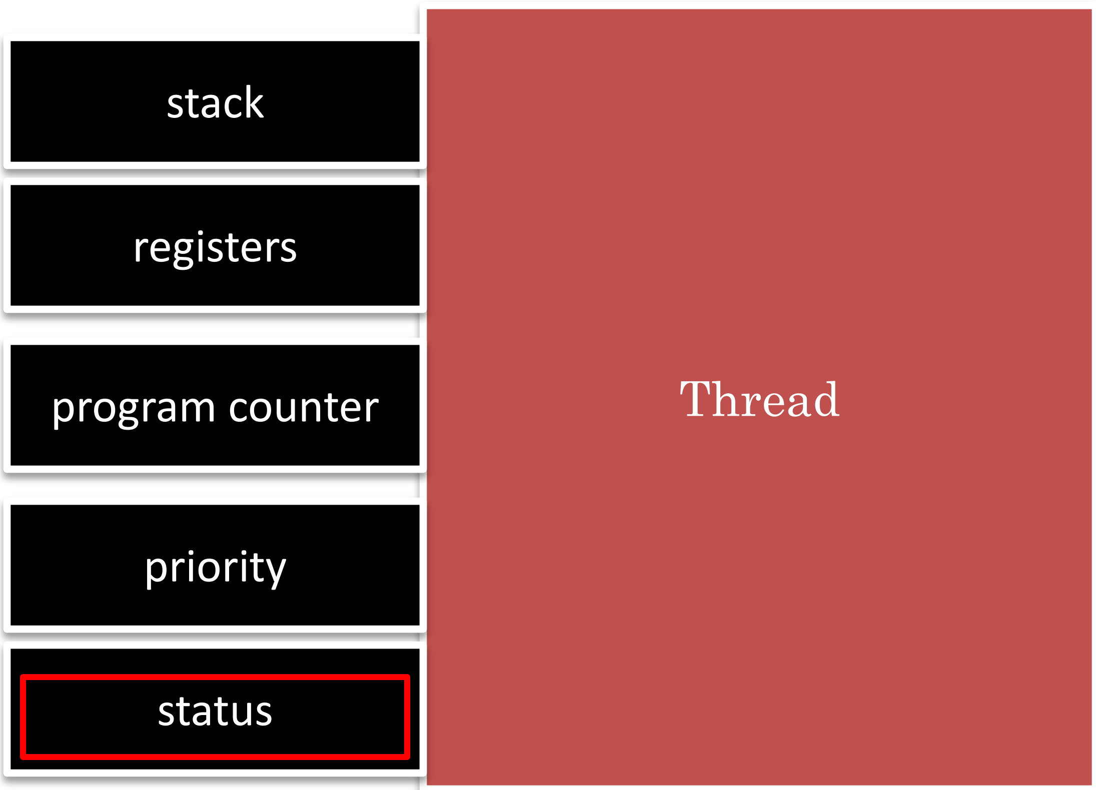
Process is execution instance of program, associated with program and memory

Thread is an independent activity, within a process, associated with a process and a stack

Thread as an Active Agent with Data Structures



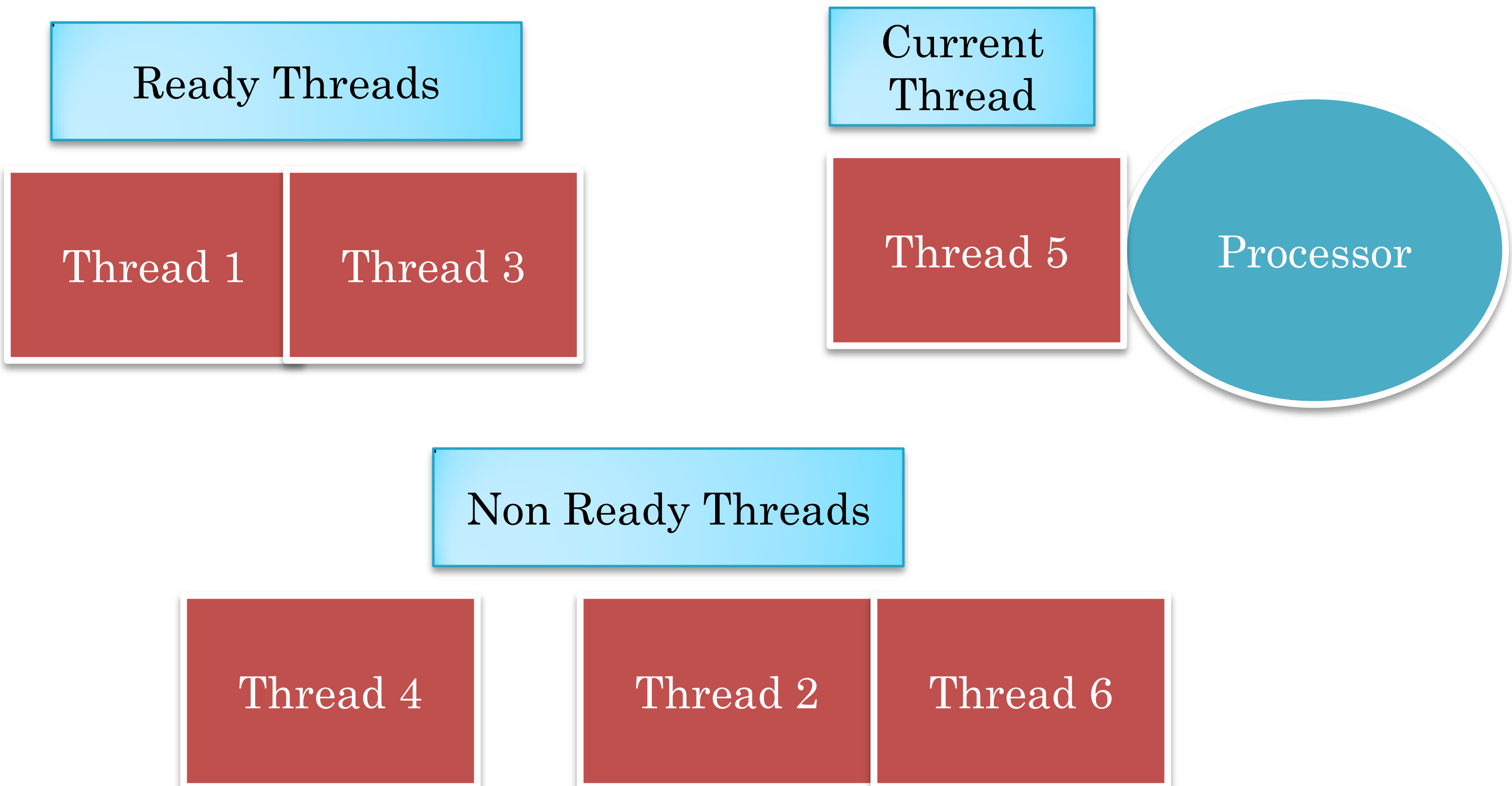
Thread as a data Structure to Implementation



To the thread implementation thread is a data structure

Thread implementation can be operating system, programming language, library

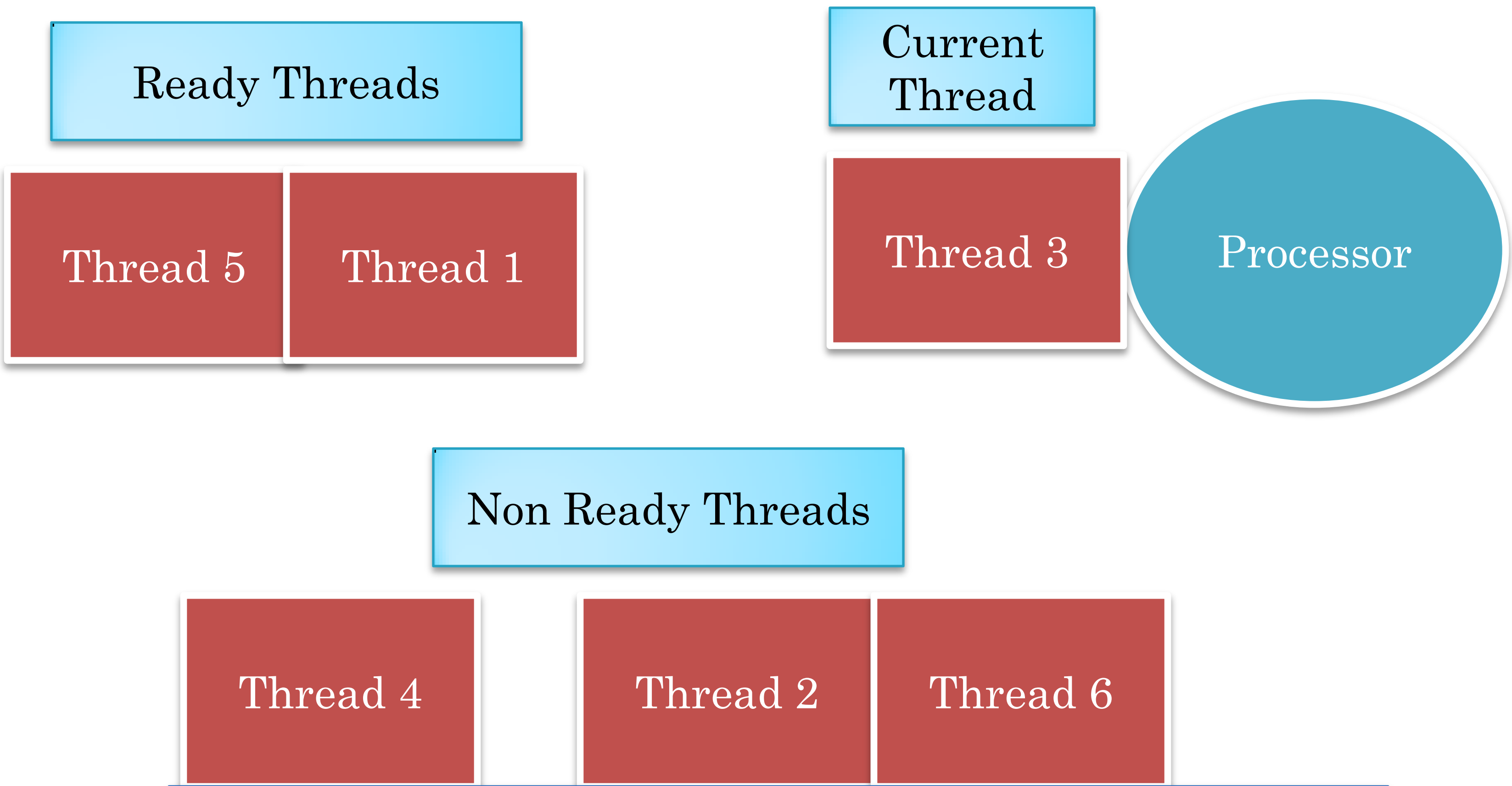
Thread as an Independent Activity



Whether current, ready, non ready captured by thread status

What triggers context switching?

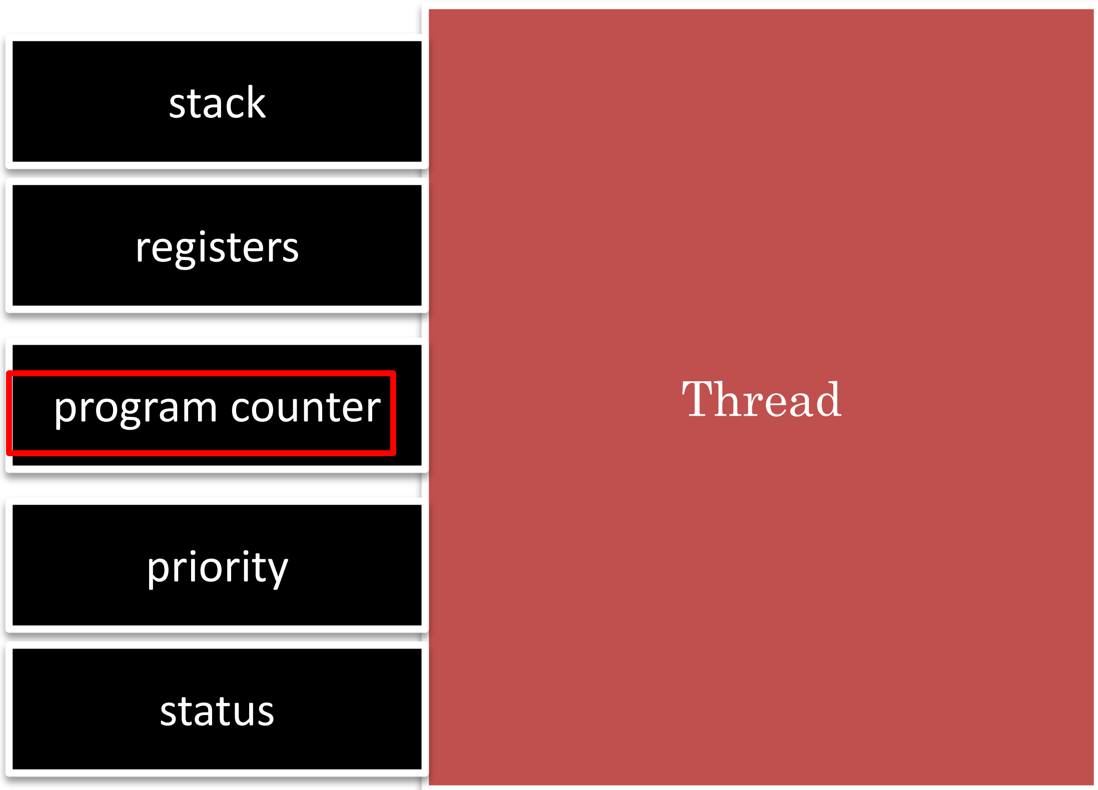
Rescheduling



Context switching occurs as a result of time quantum expiring or some other higher priority thread becoming ready

With multiple CPUs, multiple current threads, and maybe even multiple ready queues

Thread as a data Structure



Where does a thread start executing

What Does a Thread Execute?

Executes some identifiable portion of a program

Executes a method call asynchronously

Thread creator does not wait for method to terminate

Procedure Call vs. Thread Creation

```
buffer.put ("hello");
System.out.println("Put complete");
```

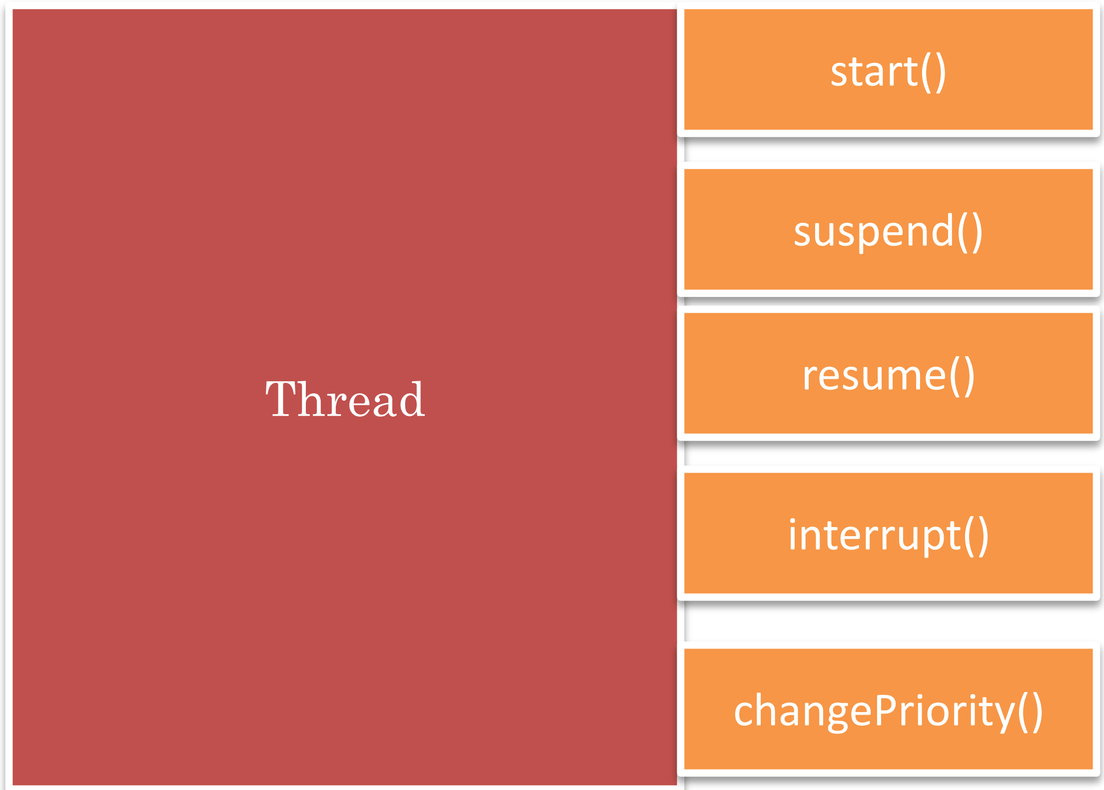
```
Thread putThread = async buffer.put ("hello");
System.out.println("Put started");
```

Not real Java code, Java syntax used to illustrate design choices

Why thread object?

What thread operations?

Thread as an Object



A thread is an object representing an independent execution of code and can be started., suspended, resumed, interrupted while sleeping, given lower/higher priority …

Procedure call vs. Thread Creation

```
Thread putThread = async buffer.put ("hello");
System.out.println("Put started");
```

Without language syntax?

Procedure call vs. Thread Creation

```
Thread putThread =  (new Thread (buffer, put, “ca va”)).start();
System.out.println(“Put started”);
```

Object address

Method address

Method parameters

Thread constructor must take varying number of arguments

Varying number of parameters to constructor /method not allowed in type safe languages

Method parameters not allowed in most OO languages

Command Object

Command Object =Embedded Operation + Parameters

execute ()

constructor (targetObject, param1, param2, …);

Provides an execute operation defined by a well-defined interface to execute some procedure

Constructor takes target object and parameters of operation as arguments

The execute operation takes no arguments

A command represents a procedure call

Java Runnable Command Object

Runnable Implementation

run()

constructor (targetObject, param1, param2, …);

```
package java.lang;
public interface Runnable {
        public void run();
}
```

Command object defined by the Runnable interface

Example Runnable Implementation

```
public class AProducer  implements Runnable {
  BoundedBuffer<String> boundedBuffer;
  String element;
  public AProducer(
     BoundedBuffer<String> aBoundedBuffer,
     String aString) {
       boundedBuffer = aBoundedBuffer;
       element = anElement;
    }
  @Override
  public void run() {
       boundedBuffer.put(element);
  }
}
```

Type parameter assigned as in List

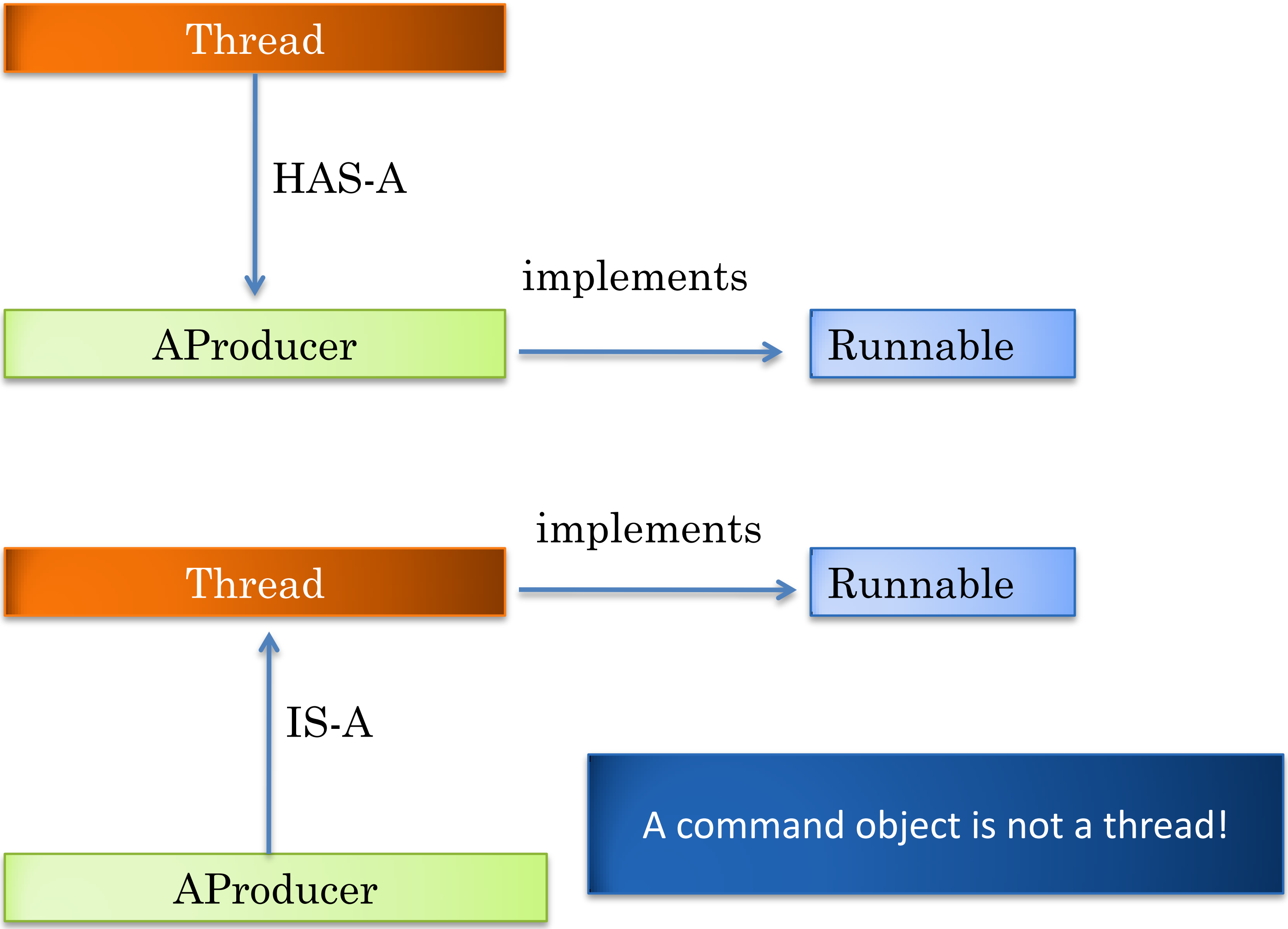
run()

constructor (targetObject, param1, param2, …);

Thread Creation

```
public class ABoundedBufferMain {
  public static void main(String[] args) {
       BoundedBuffer<String> greetings =
        new ABoundedBuffer();
       Runnable producer1 =
              new AProducer<String>(greetings, "Hello");
       Runnable producer2 =
              new AProducer<String>(greetings, "Ca Va");
       (new Thread(producer1)).start();
       (new Thread(producer2)).start();
 }
}
```

Delegation vs. Inheritance Based Thread/Command Object



Generic Elaboration

```
public class ABoundedBufferMain {
  public static void main(String[] args) {
        BoundedBuffer<String> greetings =
          new ABoundedBuffer();
        Runnable producer1 =
               new AProducer<String>(greetings, "Hello");
        Runnable producer2 =
               new AProducer<String>(greetings, "Ca Va");
        (new Thread(producer1)).start();
        (new Thread(producer2)).start();
  }
}
```

BoundedBuffer elaborated

How defined?

Generic Bounded Buffer Interface

```
public interface BoundedBuffer<ElementType> {
  void put(ElementType element);
  ElementType get();
}
```

Type parameter declared

Generic BoundedBuffer Interface

Java Generic Types (Type Parameters)

A scope name (class, interface, method) can be succeeded by a series of type parameters within angle brackets <A, B, C, …> that have the same value in all places

```
public interface I<T> {
        public void addElement (T t);
        public T elementAt (int index);
        public int size();
    }
```

A method, class, interface with type parameter is called a **generic**.

Create an **elaboration** of generic by giving actual value to type parameter

A single implementation is shared by all of its elaborations

```
I<String>  stringI;
I<Point> pointI;
```

Assigning values to type parameters is a compile time activity for type checking only

Generic Bounded Buffer Class

```
public class ABoundedBuffer<ElementType>
    implements BoundedBuffer<ElementType>{
  // data structures
  …
  void put(ElementType element) {
      …
  }
  ElementType get() {
       …
   }
}
```

Can we give them different names?

Multiple occurrences of a type variable assigned (unified to) the same value

If a class is elaborated with String then so is the interface

Why repeat type parameter in class and interface?

Generic Bounded Buffer Class

```
public class ABoundedBuffer
    implements BoundedBuffer<String>{
  // data structures
  …
  void put(String element) {
      …
  }
  ElementType get() {
       …
   }
}
```

Unparameterized class elaborating parameter of interface

May want non genric class implementing generic interface

Renaming and Multiple Type Variables

```
public interface I1 <T>{
        …
 }
```

```
public interface I2 <T>{
        …
 }
```

```
public class C<T1, T2> implements  I1 <T1>, I2<T2>{
        …
 }
```

Parameterized class may implement multiple parameterized interfaces

Generic Bounded Buffer Interface

```
public interface BoundedBuffer<ElementType> {
  void put(ElementType element);
  ElementType get();
}
```

Implementation?

A Bounded Buffer (Data Structures)

```
public class ABoundedBuffer<ElementType>
    implements BoundedBuffer<ElementType>{
      public static final int  MAX_SIZE = 10;
      Object[] buffer = new Object[MAX_SIZE];
      int size = 0;
      int nextIn = 0;
      int nextOut = 0;
```

Convention: Class A<T> implements interface <T> containing public methods of A

Bounded Buffer (Methods)

```
public void put(ElementType element) {
      if (size >= MAX_SIZE) {
        return;
      }
      buffer[nextIn] = element;
      nextIn = (nextIn + 1) % MAX_SIZE;
      size++;
}
public ElementType get() {
      if (size == 0) {
        return null;
      }
      ElementType retVal = (ElementType) buffer[nextOut];
      nextOut = (nextOut + 1) % MAX_SIZE;
      size--;

}
}
```

Requires polling to wait for non empty and empty slots

Thread safe?

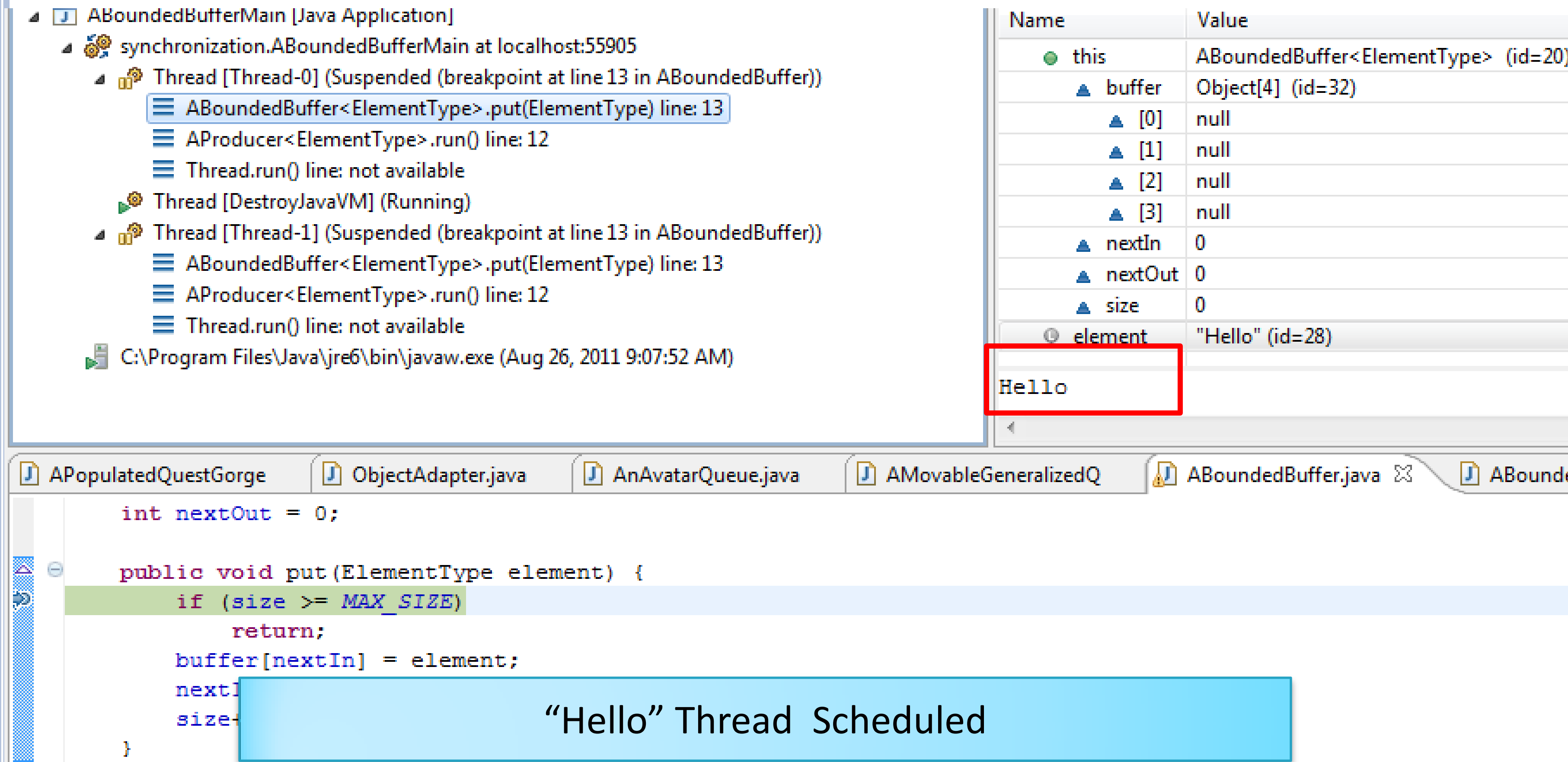
Will it work if multiple threads access it

Example Threads

```
public class ABoundedBufferMain {
  public static void main(String[] args) {
       BoundedBuffer<String> greetings =
        new ABoundedBuffer();
       Runnable producer1 =
             new AProducer<String>(greetings, "Hello");
       Runnable producer2 =
             new AProducer<String>(greetings, "Ca Va");
       (new Thread(producer1)).start();
       (new Thread(producer2)).start();
 }
}
```

"Hello" and "Ca Va" threads created

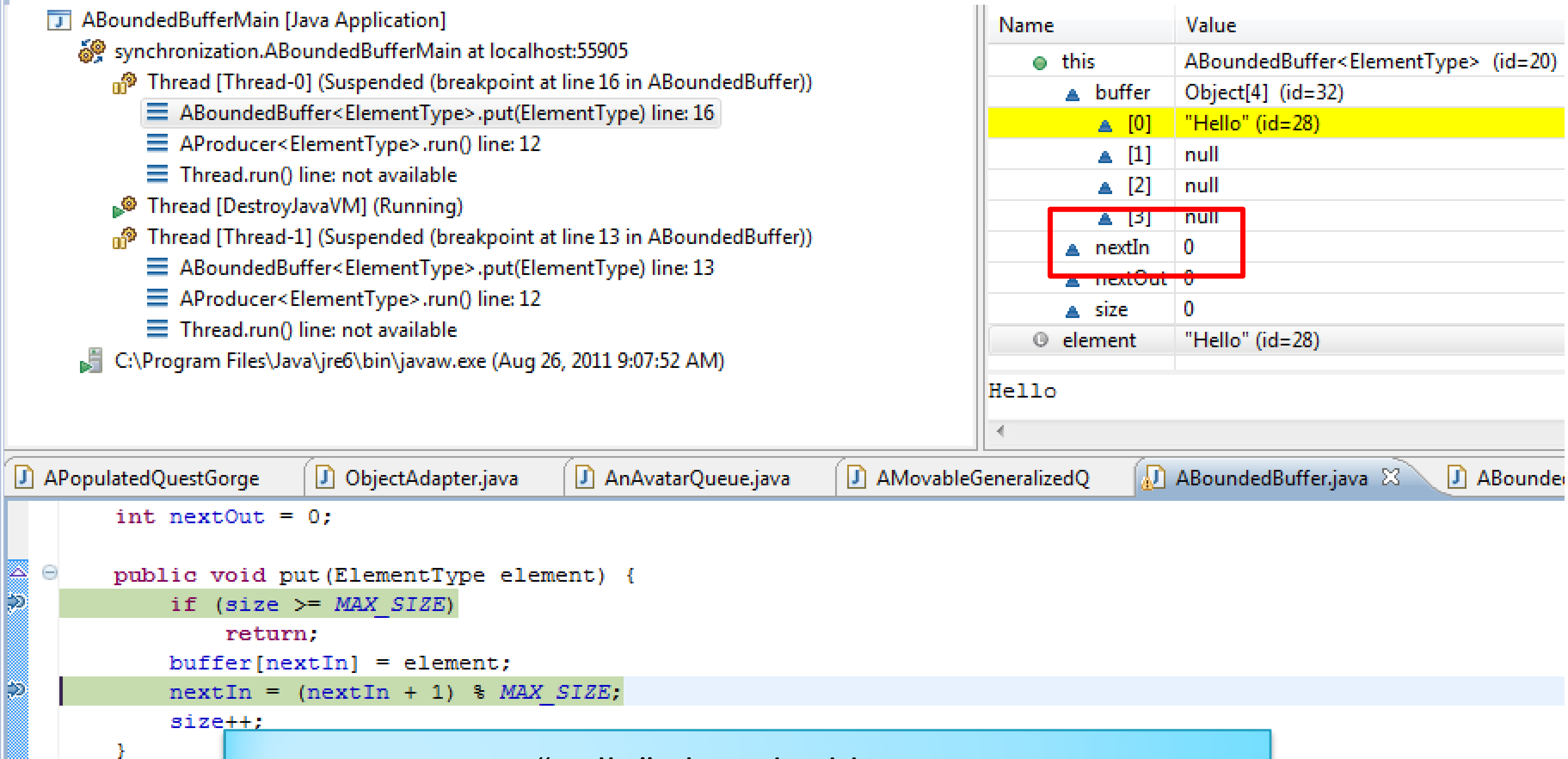
“Hello” Thread



“Hello” Thread Scheduled

“Hello” Thread Starts Executing put()

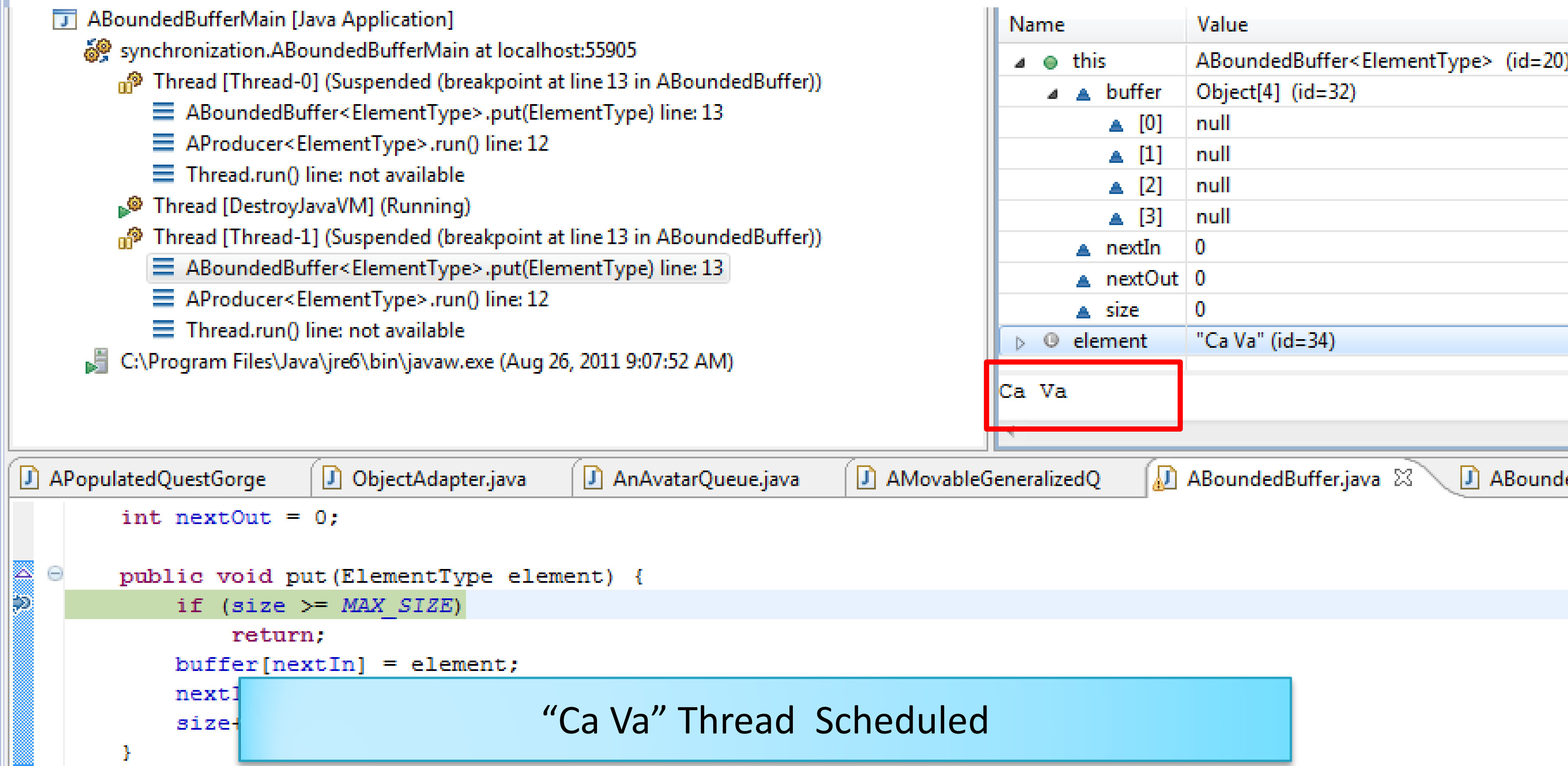
"Hello" Thread Proceeds



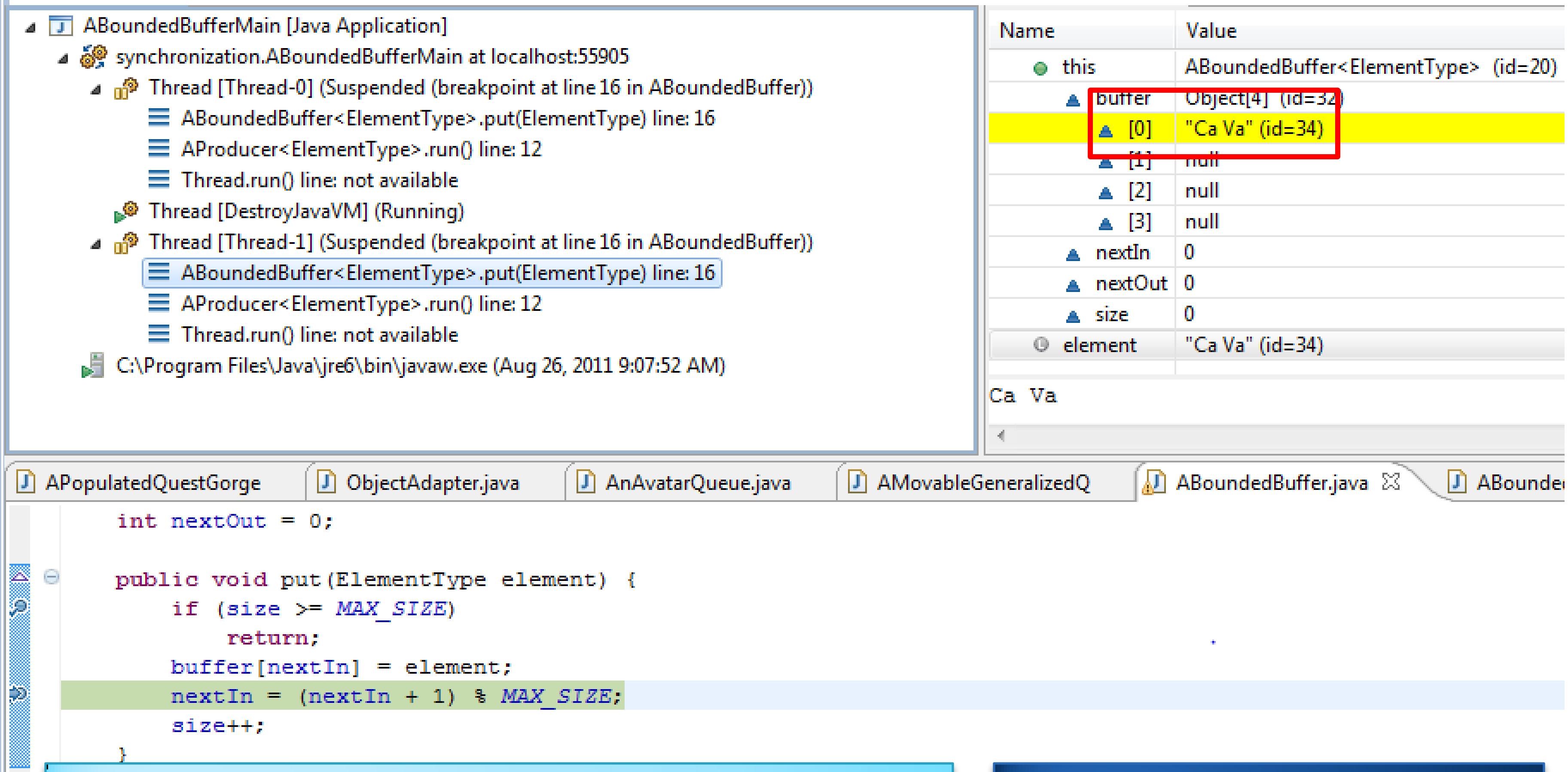
"Hello" Thread Adds Item

nextIn not incremented before rescheduling happens

“Ca Va” Thread


ABoundedBufferMain [Java Application]
synchronization.ABoundedBufferMain at localhost:55905
Thread [Thread-0] (Suspended (breakpoint at line 13 in ABoundedBuffer))
ABoundedBuffer<ElementType>.put(ElementType) line: 13
AProducer<ElementType>.run() line: 12
Thread.run() line: not available
Thread [DestroyJavaVM] (Running)
Thread [Thread-1] (Suspended (breakpoint at line 13 in ABoundedBuffer))
ABoundedBuffer<ElementType>.put(ElementType) line: 13
AProducer<ElementType>.run() line: 12
Thread.run() line: not available
C:\Program Files\Java\jre6\bin\javaw.exe (Aug 26, 2011 9:07:52 AM)
Name
Value
this
ABoundedBuffer<ElementType> (id=20)
buffer
Object[4] (id=32)
[0]
null
[1]
null
[2]
null
[3]
null
nextIn
0
nextOut
0
size
0
element
"Ca Va" (id=34)
Ca Va
APopulatedQuestGorge
ObjectAdapter.java
AnAvatarQueue.java
AMovableGeneralizedQ
ABoundedBuffer.java
ABounde
int nextOut = 0;
public void put(ElementType element) {
if (size >= MAX_SIZE)
return;
buffer[nextIn] = element;
nextI
size+
}
“Ca Va” Thread Scheduled

Context Switch to "Ca Va" Thread



Previous greeting overwritten

Non deterministic program

Program behavior depends on when rescheduling occurs

Critical Section

Critical Section

Code that can be executed only if certain conditions are met by other threads

Other threads should not be executing the same critical section

Other threads should not be executing other critical sections that access the same data

Critical Section (Review)

Critical Section

Code that can be executed only if certain conditions are met by other threads

Other threads should not be executing the same critical section

Other threads should not be executing other critical sections that access the same data

CRITICAL SECTION

```
lock.lock()
```

Critical Section

```
lock.unlock()
```

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

Critical Section (Review)

```
lock.lock()
```

Critical Section

```
lock.unlock()
```

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

Use of Lock

```
public class ALockingBoundedBuffer<ElementType>
             extends ABoundedBuffer<ElementType>{
  Lock lock;
  public ALockingBoundedBuffer(Lock aLock) {
       lock = aLock;
  }
  public void put(ElementType element) {
       lock.lock();
       super.put(element);
       lock.unlock();
  }
  public  ElementType get() {
       lock.lock();
       ElementType retVal = super.get();
       lock.unlock();
       return retVal;
  }
}
```

How to implement lock?

Disable Context Switching

```
public class AContextSwitchingLock implements Lock{
   int originalInterrupts;
   public void lock() {
        originalInterrupts = System.disableInterrupts();
   }
   public void unlock() {
       System.restoreInterrupts(originalInterrupts);
   }
}
```

Context switching occurs as a result of a blocking call or an interrupt

Timer interrupt can cause time quantum to expire

Input/disk/network interrupt can cause higher priority thread to becoming eligible

Disable interrupts and make no blocking call in critical section

Pros and Cons

Coarse grained: stops all threads, even those that do not access same or any critical section

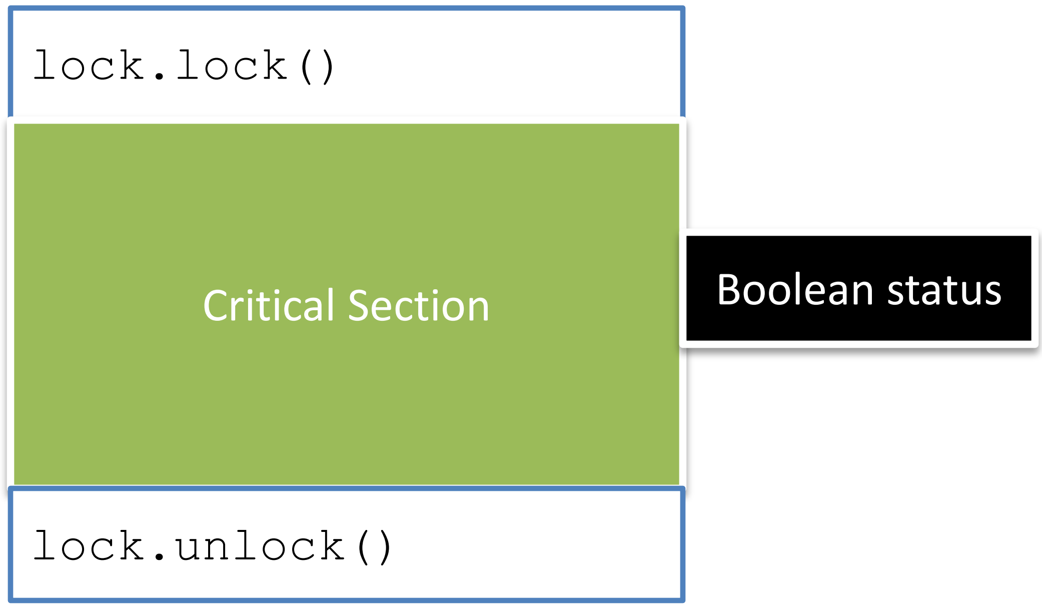
Insecure: a thread can cheat and hog processor time

Simple

Needed to implement critical sections in operating system

Need critical sections to implement critical sections!

Fine-Grained Critical Section



Boolean Object Capturing Lock Status

```
public class ABooleanObject implements BooleanObject{
  boolean value;
  public boolean get() {
       return value;
  }
  public void set(boolean newVal) {
       value = newVal;
  }
}
```

boolean is not an object

Boolean is not an mutable

Busy Waiting Lock

```
public class APollingLock implements Lock{
  BooleanObject booleanObject = new ABooleanObject();
  public void lock() {
       while (booleanObject.get()) {
         ; // do nothing
       }
       booleanObject.set(true);
  }
  public void unlock() {
       booleanObject.set(false);
  }
}
```

Test of lock

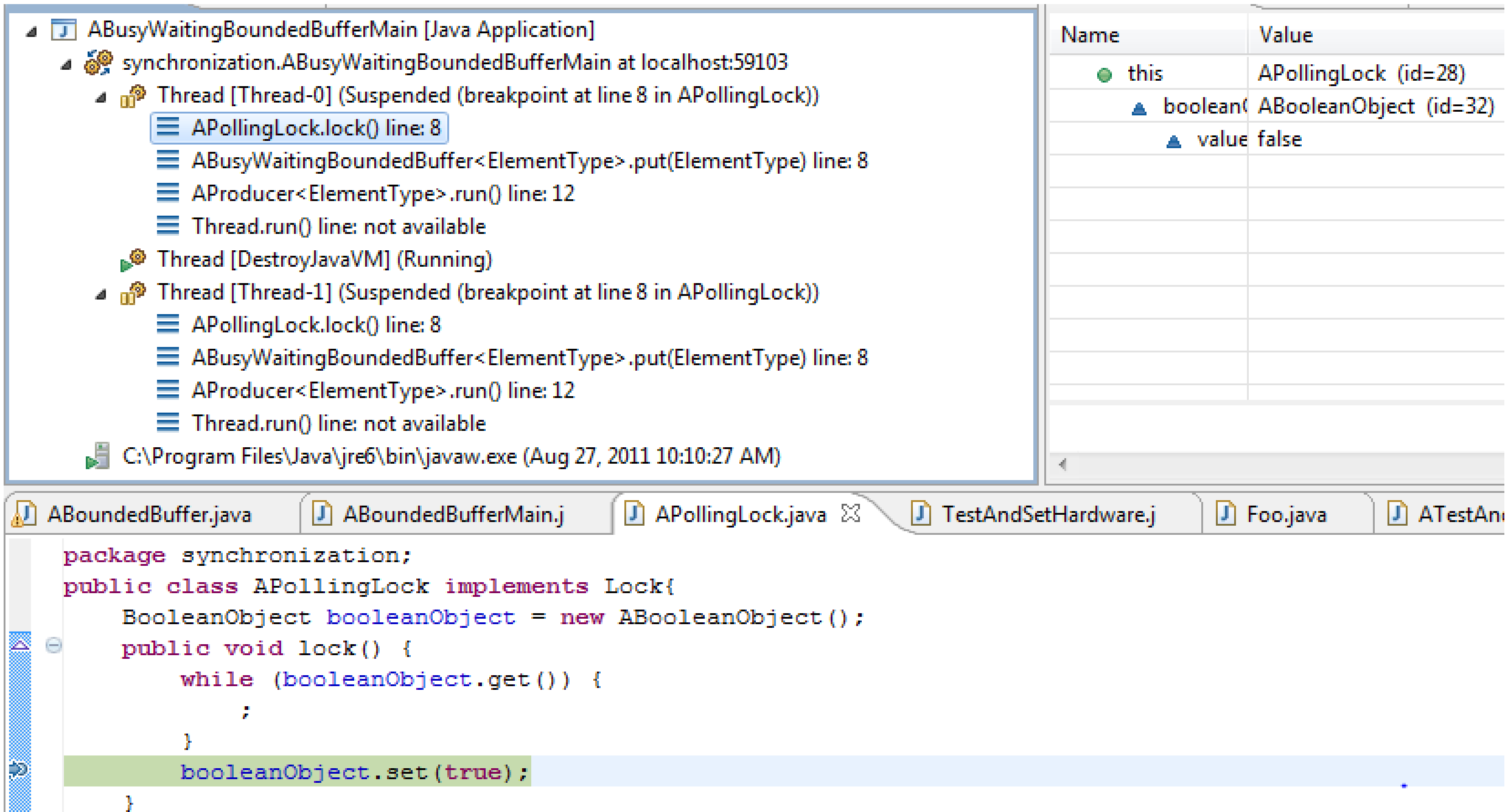
Set of lock

Does this work?

Test and set not done atomically

Context switch can occur between test and set

Thread 1 Gets Lock



Thread 1 gets lock and context switches

Thread 2 Also Gets Lock

ABusyWaitingBoundedBufferMain [Java Application]
synchronization.ABusyWaitingBoundedBufferMain at localhost:59103
Thread [Thread-0] (Suspended (breakpoint at line 8 in APollingLock))
APollingLock.lock() line: 8
ABusyWaitingBoundedBuffer<ElementType>.put(ElementType) line: 8
AProducer<ElementType>.run() line: 12
Thread.run() line: not available
Thread [DestroyJavaVM] (Running)
Thread [Thread-1] (Suspended (breakpoint at line 8 in APollingLock))
APollingLock.lock() line: 8
ABusyWaitingBoundedBuffer<ElementType>.put(ElementType) line: 8
AProducer<ElementType>.run() line: 12
Thread.run() line: not available
C:\Program Files\Java\jre6\bin\javaw.exe (Aug 27, 2011 10:10:27 AM)

Name	Value
this	APollingLock (id=28)
boolean(	ABooleanObject (id=32)
value	false

ABoundedBuffer.java | ABoundedBufferMain.j | APollingLock.java | TestAndSetHardware.j | Foo.java | ATestAnd

```
package synchronization;
public class APollingLock implements Lock{
    BooleanObject booleanObject = new ABooleanObject();
    public void lock() {
        while (booleanObject.get()) {
            ;
        }
        booleanObject.set(true);
    }
```

Thread 2 gets lock

Busy Waiting Lock

```
public class APollingLock implements Lock{
   BooleanObject booleanObject = new ABooleanObject();
   public void lock() {
        while (booleanObject.get()) {
          ; // busy waiting
        }
        booleanObject.set(true);
   }
   public void unlock() {
        booleanObject.set(false);
   }
}
```

Test of lock

Set of lock

Test and set not done atomically

Fix how?

Context switch can occur between test and set

Test and Set Hardware Instruction

```
public class TestAndSetHardware {
  public static atomic boolean
    testAndSet(BooleanObject aBooleanObject) {
      boolean retVal = aBooleanObject.get(); // test
      aBooleanObject.set(true); // set
      return retVal; // return tested value
  }
}
```

Function with side effects!

At the end of instruction, boolean object (memory slot) is always true

returns false if object was false at start of instruction

A Test and Set Lock

```
public class ATestAndSetLock implements Lock{
  BooleanObject booleanObject;
  public void lock() {
     while (TestAndSetHardware.testAndSet(booleanObject)) {
              ;
     }
  }
  public void unlock() {
       booleanObject.set(false);
  }
}
```

Also called spin locks, as tester is spinning

Busy Waiting Pros and Cons

Busy waiting: polling wastes computer resources

Higher priority polling thread would not allow lower priority thread in critical section to release lock

Needed to implement critical sections in multiprocessor operating system

Disabling interrupts on one processor does not disable on others

Fine-grained approach with out disadvantages?

SEMAPHORE

```
public class ASemaphore implements Semaphore{
  Queue<Thread> queue = new AQueue<Thread>();
  int count;
  public ASemaphore(int initialCount) {
       count = initialCount;
  }
  public void semWait() {
       int originalInterrupts = System.disableInterrupts();
       count--;
       if (count < 0) {
             Thread currentThread = Thread.currentThread();
             currentThread.setStatus(Thread.WAIT);
             queue.add(currentThread);
       }
       System.restoreInterrupts(originalInterrupts);
  }
```

Count and queue manipulation must be done atomically

Disable and restore interrupts to ensure atomicity

Semaphore (Contd)

```
public void semSignal() {
     int originalInterrupts = System.disableInterrupts();
     count++;
     if (count <= 0) {
       Thread  nextThread = queue.remove();
       nextThread.setStatus(Thread.READY);
       Thread.reschedAndRestoreInterrupts(
            originalInterrupts);
     } else{
       System.restoreInterrupts(originalInterrupts);
  }
 }
```

If count is positive, indicates how many semWait() can be done without blocking

If count is negative, its absolute value indicates queue size

Semaphore-Based Lock

```
public class ASemaphoreLock implements Lock {
  // let first thread go
  Semaphore semaphore = new ASemaphore(1);
  public void lock() {
       semaphore.semWait();
  }
  public void unlock() {
       semaphore.semSignal();
  }
}
```

Semaphore Pros and Cons

Fine-grained locking that works and is efficient

Like other schemes discussed so far, requires explicit lock and unlock

Programmer overhead in creating and using lock

Programmer may forget to lock

Worse, may forget to unlock

Program reader must look at code to find critical sections

Manual Use of Lock

```
public class ALockingBoundedBuffer<ElementType>
             extends ABoundedBuffer<ElementType>{
  Lock lock;
  public ALockingBoundedBuffer(Lock aLock) {
       lock = aLock;
  }
  public void put(ElementType element) {
       lock.lock();
       super.put(element);
       lock.unlock();
  }
  public  ElementType get() {
       lock.lock();
       ElementType retVal = super.get();
       lock.unlock();
       return retVal;
  }
}
```

Create declarative rather than procedural solution

Declarative vs. Procedural Primitive

```
public class ALockingBoundedBuffer<ElementType>
             extends ABoundedBuffer<ElementType>{
  Lock lock;
  public ALockingBoundedBuffer(Lock aLock) {
      lock = aLock;
  }
  public void put(ElementType element) {
      lock.lock();
      super.put(element);
      lock.unlock();
  }
  public  ElementType get() {
      lock.lock();
      ElementType retVal = s
      lock.unlock();
      return retVal;
  }
}
```

Declarative: Directive to system by giving declaration specifying *what*

Procedural: Directive to system by executing statements specifying *how*

A la Regular expressions vs. Finite State Automata

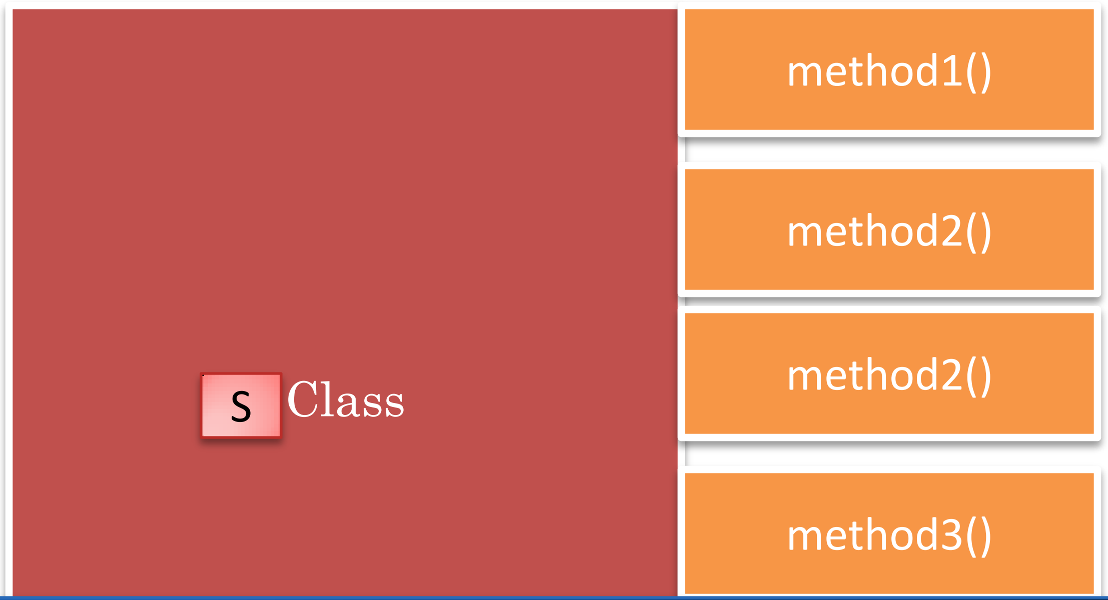
High-Level Support

Critical Section

Other threads should not be executing other critical sections that access the same data

Connect lock to shared data and methods that manipulate them

Class with All Atomic Methods



When declaring a class, provide a label (e.g. synchronized) that says that on an instance a method cannot be executed if the same or another method is executing on that instance

Such a class is a monitor class, and its instance a monitor

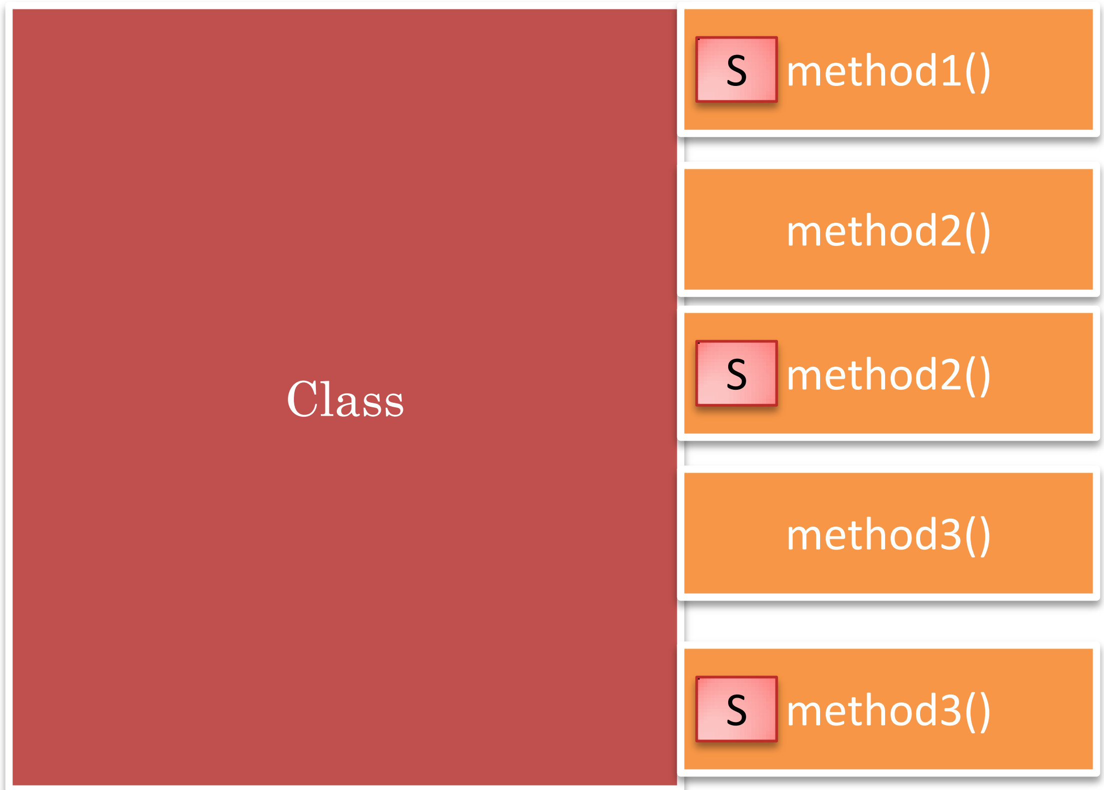
Not all classes have critical sections, hence labeling can prevent unnecessary code and data structures in such non monitor classes

Labeling a Class as a Monitor

```
public synchronized ABoundedBuffer<ElementType>
    implements BoundedBuffer<ElementType>{
      public static final int  MAX_SIZE = 10;
      Object[] buffer = new Object[MAX_SIZE];
      int size = 0;
      int nextIn = 0;
      int nextOut = 0;
```

Some methods of the class may not be critical sections

Atomic vs Non Atomic Methods



When declaring a method, provide a label (e.g. synchronized). A single synchronized method can execute at one time on an instance

Labeling Methods as Atomic

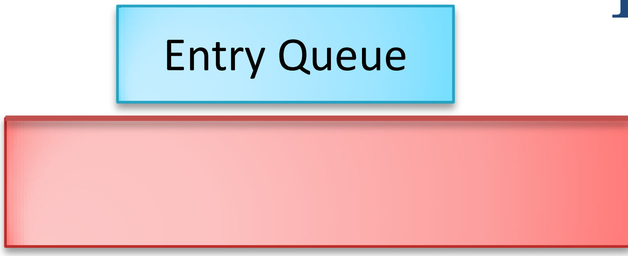
```
public synchronized void put(ElementType element) {
    if (size >= MAX_SIZE) {
      return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
}
public synchronized ElementType get() {
    if
    }
    El                                         t];
    ne
    si
    re
}
}
```

A lock associated with class instance is automatically obtained on entry to the marked procedure and released on its exit

Such a method is classically called an **entry procedure**

Java calls it a **synchronized method**, we will call it an **atomic method**

Entry Queue




```
public synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
     return;
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
      return null;
   }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

GET CALLED

Entry Queue

Cons. 1

```
public synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
     return;
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
      return null;
   }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Get Proceeds and Put Called

Entry Queue

Prod. 1

Cons. 1

```java
blic synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
     return;
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
      return null;
   }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Producer in Entry Queue

Entry Queue

Prod. 1

Cons. 1

```
blic synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
     return;
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
      return null;
   }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Put Called Again

Entry Queue

Prod. 1

Prod. 2

Cons. 1

```
blic synchronized void put(
   ElementType element) {
  if (size >= MAX_SIZE) {
     return;
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
      return null;
   }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Producer in Entry Queue

Entry Queue

Prod. 2 Prod. 1

Cons. 1

```
blic synchronized void put(
  ElementType element) {
  if (size >= MAX_SIZE) {
     return;
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
      return null;
   }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Consumer About To Return

Entry Queue

Prod. 2 Prod. 1

Cons. 1

```
blic synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
      return;
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
        return null;
   }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Consumer Leaves

Entry Queue

Prod. 2 Prod. 1

Monitor unlocked

```
public synchronized void put(
     ElementType element) {
  if (size >= MAX_SIZE) {
     return;
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
      return null;
   }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

FIFO Service of Entry Queue

Entry Queue

Prod. 2

Prod. 1

```
blic synchronized void put(
  ElementType element) {
  if (size >= MAX_SIZE) {
     return;
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
      return null;
   }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Monitor Pros and Cons

Higher-level declarative solution

No programmer overhead in creating and using lock

No danger of forgetting to unlock

Can simply look at method headers to determine critical sections

Can forget to label method

Does not provide (general) thread synchronization

Synchronization vs. Mutual Exclusion

Thread general synchronization (Thread coordination): a thread waiting for another thread to do something

Thread mutual exclusion: : a thread waiting for another thread to leave a critical section

Threads are “competing” rather than “cooperating”

Thread synchronization: general synchronization other than mutual exclusion

Threads are “cooperating” rather than “competing”

Thread Synchronization Need

```
public synchronized void put(El
    if (size >= MAX_SIZE) {
      return;
    }
    buffer[nextIn] = element;
    nextIn = (nextIn + 1) % MAX_SIZE;
    size++;
}
public synchronized ElementTyp
    if (size == 0) {
      return null;
    }
    ElementType retVal = (ElementType) buffer[nextOut];
    nextOut = (nextOut + 1) % MAX_SIZE;

}
}
```

Should wait for non full buffer

Should wait for non empty buffer

Semaphores? How many?

Monitors do not support general thread synchronization

Thread Synchronization with Semaphores

```
public synchronized void put(ElementType element) {
      nonFull.semWait();
      buffer[nextIn] = element;
      nextIn = (nextIn + 1) % MAX_SIZE;
      size++;
      nonEmpty.semSignal();
  }
  public synchronized ElementType get() {
      nonEmpty.semWait();
      ElementType retVal = (ElementType) buffer[nextOut];
      nextOut = (nextOut + 1) % MAX_SIZE;
      size--;
      nonFull.semSignal();
      return retVal;
  }
}
```

Initial counts of semaphores?

Bounded Buffer (Data Structures)

```
public class ABoundedBuffer<ElementType>
    implements BoundedBuffer<ElementType>{
      public static final int  MAX_SIZE = 10;
      Object[] buffer = new Object[MAX_SIZE];
      int size = 0;
      int nextIn = 0;
      int nextOut = 0;
      Semaphore nonFull = new ASemaphore(MAX_SIZE);
      Semaphore nonEmpty = new ASemaphore(0);
```

should be able to deposit MAX_SIZE buffers without waiting

General synchronization solution: Monitors + Semaphores?

Semaphores too Heavyweight

```
public synchronized void put(E
      nonFull.semWait();
      buffer[nextIn] = element;
      nextIn = (nextIn + 1) % MAX_SIZE;
      size++;
      nonEmpty.semSignal();
  }
  public synchronized ElementType get() {
      nonEmpty.semWait();
      ElementType retVal = (Elem
      nextOut = (nextOut + 1) %
      size--;
      nonFull.semSignal();
      return retVal;
  }
}
```

Semaphore takes steps to ensure atomic execution of semaphore operations

But method has already taken steps to ensure atomicity

Unnecessary expensive system calls to disable context switching

Condition (Synchronizer)

```
public class AConditionSynchronizer
      implements ConditionSynchronizer{
  Queue<Thread> queue = new AQueue<Thread>();
  public void condWait() {
      Thread currentThread = Thread.currentThread();
      currentThread.setStatus(Thread.WAIT);
      queue.add(currentThread);
  }
  public void condSignal() {
        Thread  nextThread = queue.remove();
        nextThread.setStatus(Thread.READY);
        Thread.reschedAndRestoreInterrupts(
             originalInterrupts);
```

Non atomic operations – no disabling of context switching

int count is only one kind of condition for synchronization

No integer count

Synchronization condition checked by caller

Policy and Mechanism Tied In

```
  public synchronized void put(ElementType element) {
        nonFull.semWait();
        buffer[nextIn] = element;
        nextIn = (nextIn + 1) % MAX_SIZE;
        size++;
        nonEmpty.semSignal();
  }
  public synchronized ElementType get() {
        nonEmpty.semWait();
        ElementType retVal = (ElementType) buffer[nextOut];
        nextOut = (nextOut + 1) % MAX_SIZE;
        size--;
        nonFull.semSignal();
        return retVal;
  }
}
```

Policy and Mechanism Orthogonal

```
  public synchronized void put(ElementType element) {
      if (size >= MAX_SIZE) {
        nonFull.condWait();
      }
      buffer[nextIn] = element;
      nextIn = (nextIn + 1) % M
      size++;
      nonEmpty.condSignal();
  }
  public synchronized ElementType get() {
      if (size == 0) {
        return nonEmpty.condWai
      }
      ElementType retVal = (Ele
      nextOut = (nextOut + 1) % MAX_SIZE;
      size--;
      nonFull.condSignal()
      return retVal;
  }
}
```

Must check programmer-defined boolean expression before waiting

Policy for synchronizing and mechanism for blocking are orthogonal

Could we put check outside semaphore also

Lightweight Semaphore

```
public class ASemaphore implements Semaphore{
  Queue<Thread> queue = new AQueue<Thread>();
  public void semWait() {
       int originalInterrupts = System.disableInterrupts();
       Thread currentThread = Thread.currentThread();
       currentThread.setStatus(Thread.WAIT);
       queue.add(currentThread);
       System.restoreInterrupts(originalInterrupts);
  }
  public void semSignal() {
       int originalInterrupts = System.disableInterrupts();
         Thread  nextThread = queue.remove();
         nextThread.setStatus(Thread.READY);
         Thread.reschedAndRestoreInterrupts(
               originalInterrupts);
  }
 }
```

Like regular semaphore, except no count

Like condition, except there is atomicity implementation

Lighter Weight Semaphore Use

```
  public void put(ElementType element) {
       if (size >= MAX_SIZE) {
         nonFull.semWait();
       }
       buffer[nextIn] = element;
       nextIn = (nextIn + 1) % MAX_SIZ
       size++;
       nonEmpty.semSignal();
  }
  public ElementType get() {
       if (size == 0) {
         return nonEmpty.semWait();
       }
       ElementType retVal = (ElementType) buffer[nextOut];
       nextOut = (nextOut + 1) % MAX_SIZE;
       size--;
       nonFull.semSignal()
       return retVal;
  }
}
```

Test of lock

Enqueuing

Test and enqueing are not atomic

Synchronizer Operations in Atomic Methods

```
public synchronized void put(ElementType element) {
     if (size >= MAX_SIZE) {
       nonFul condWait();
     }
     buffer[nextIn] = element;
     nextIn = (nextIn + 1) % M
     size++;
     nonEmpty ondSignal();
}
public synchronized ElementTyp
     if (size == 0) {
       return nonEmpty.condWai
     }
     ElementType retVal = (Ele
     nextOut = (nextOut + 1) % MAX_SIZE;
     size--;
     nonFull.condSignal()
     return retVal;
}
}
```

Blocking/unblocking operations should be done atomically

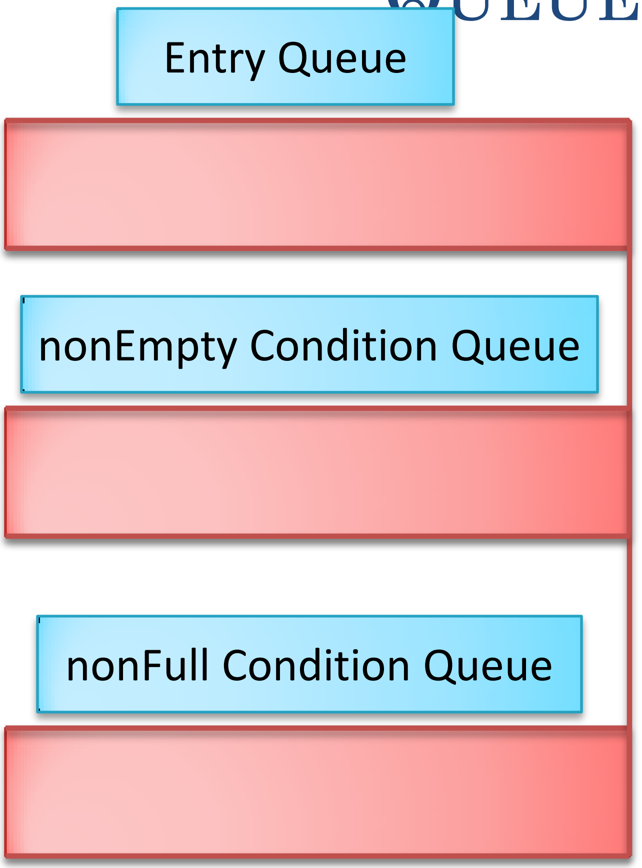
Conditions for blocking/unblocking should also be checked atomically with blocking/unblocking operations

Force blocking/unblocking operations to be done only in atomic methods, which can then check and set arbitrary conditions atomically

Bounded Buffer with Condition (Synchronizers): Data Structures

```
public class ABoundedBuffer<ElementType>
    implements BoundedBuffer<ElementType>{
       public static final int  MAX_SIZE = 10;
       Object[] buffer = new Object[MAX_SIZE];
       int size = 0;
       int nextIn = 0;
       int nextOut = 0;
       ConditionSynchronizer nonFull =
          new AConditionSynchronizer();
       ConditionSynchronizer nonEmpty =
          new AConditionSynchronizer();
```

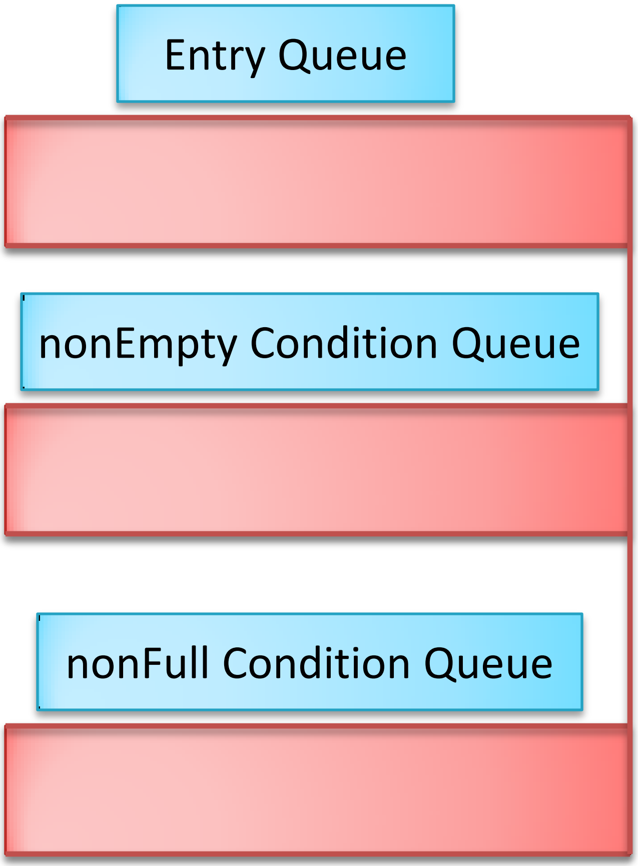
Queues with Conditions




```java
public synchronized void put(
      ElementType element) {
   if (size >= MAX_SIZE) {
      nonFull.condWait();
   }
   buffer[nextIn] = element;
   nextIn = (nextIn + 1) % MAX_SIZE;
   size++;
   nonEmpty.condSignal();
}
public synchronized ElementType get() {
   if (size == 0) {
      nonEmpty.condWait();
   }
   ElementType retVal =
      (ElementType) buffer[nextOut];
   nextOut = (nextOut + 1) % MAX_SIZE;
   size--;
   nonFull.condSignal()
   return retVal;
}
```

Get Called

Cons. 1




```
public synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Get Proceeds and Put Called

Entry Queue

nonEmpty Condition Queue

nonFull Condition Queue

Prod. 1

Cons. 1

```
blic synchronized void put(
  ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Producer in Entry Queue

Entry Queue

Prod. 1

nonEmpty Condition Queue

nonFull Condition Queue

Cons. 1

```
blic synchronized void put(
   ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Get Called Again

Entry Queue

Cons. 2 | Prod. 1

nonEmpty Condition Queue

nonFull Condition Queue

Cons. 1

```
blic synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Consumer Waits for Condition

Entry Queue

Cons. 2 Prod. 1

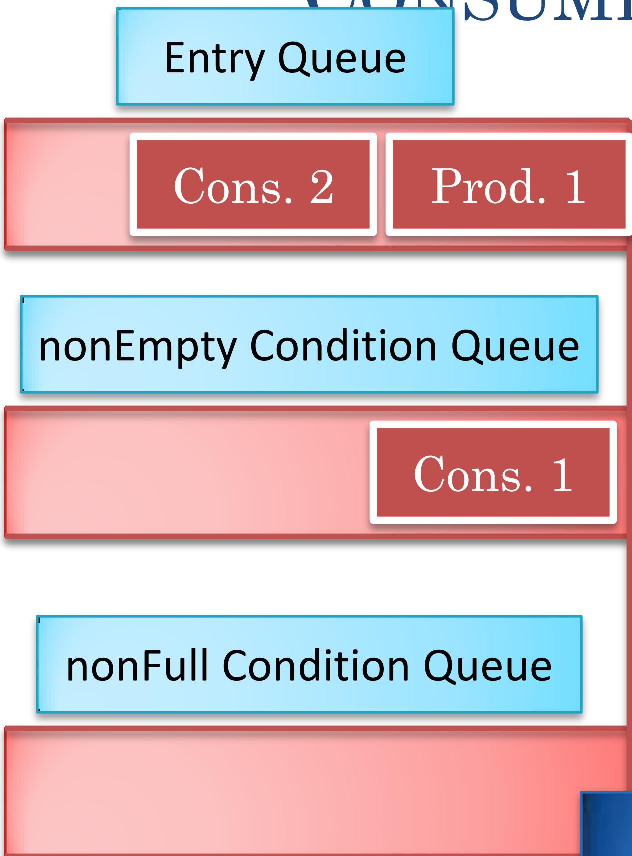
nonEmpty Condition Queue

nonFull Condition Queue

Cons. 1

```
blic synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Consumer Unlocks Monitor



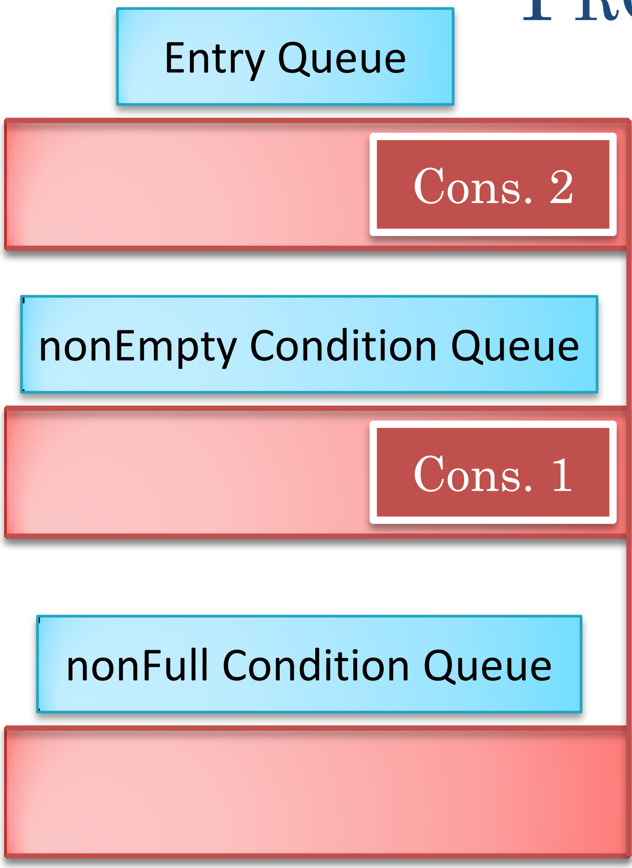

```
public synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
```

Before waiting , class invariant should be restored

Class invariant is a condition that is true before and after each public method in the class is called

E.g.: size should indicate the number of filled slots

Producer Enters



Prod. 1

```
blic synchronized void put(
  ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Producer Signals, Consumer Dequeued

Entry Queue

Cons. 2

nonEmpty Condition Queue

nonFull Condition Queue

Cons. 1

Prod. 1

```
blic synchronized void put(
   ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType ret
    (ElementType)
  nextOut = (next
  size--;
  nonFull.condSig
  return retVal;
}
```

Producer not exited

Where does consumer go next?

Signaled Thread in Monitor?

Entry Queue

Cons. 2

nonEmpty Condition Queue

nonFull Condition Queue

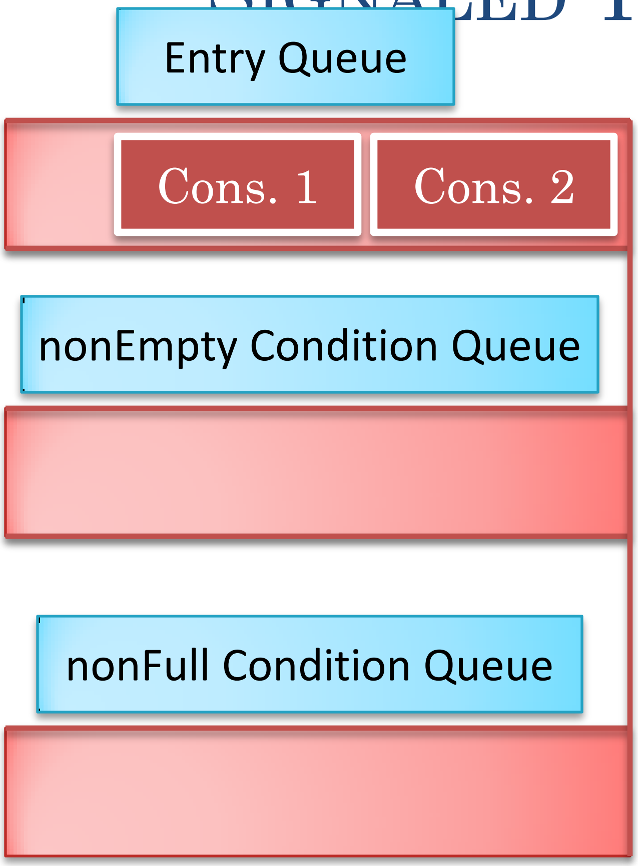
Prod. 1

Cons. 1

```
blic synchronized void put(
  ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (next
  size--;
  nonFull.condSig
  return retVal;
}
```

Violates Monitor Definition

Signaled Thread in Entry Queue



Prod. 1

```
blic synchronized void put(
  ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size-
  nonFu
  retur
}
```

Cons. 1 will get the buffer first

Starvation possible

Signaled Thread in Urgent Queue

Entry Queue

Cons. 2

nonEmpty Condition Queue

nonFull Condition Queue

Urgent Queue

Cons. 1

Prod. 1

```
blic synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++
  nonEmp
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementTy
    (Elemen
  nextOut =
  size--;
  n
  r
}
```

```
nonEmpty.condSignal();
get();
```

Signaled thread in urgent Q

Urgent Q serviced before Entry Q

Condition for Entering May Not Exist Later

Signaler may violate condition or previously signaled thread may violate it

Signaled Thread in Monitor and Signaling T Urgent Queue

Entry Queue

Cons. 2

nonEmpty Condition Queue

nonFull Condition Queue

Urgent Queue

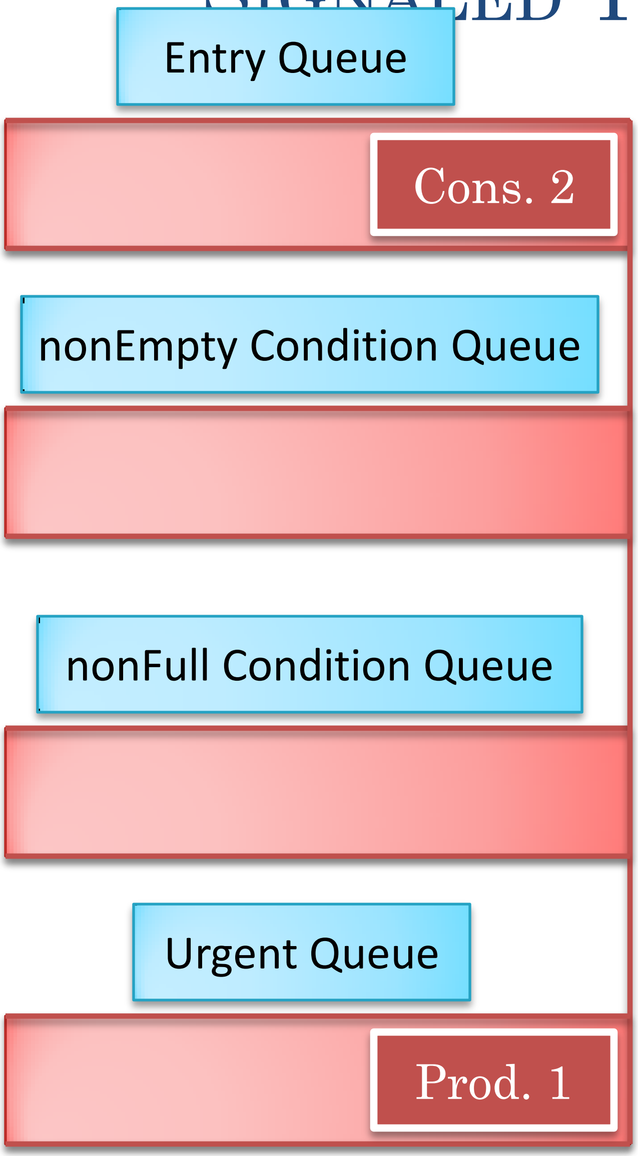
Prod. 1

```
blic synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpt
}
public sy
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Cons. 1

Signaling thread must restore monitor invariant before signaling

Signaled Thread About To Leave



Cons. 1

```
blic synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Urgent Queue Serviced First

Entry Queue

Cons. 2

nonEmpty Condition Queue

nonFull Condition Queue

Urgent Queue

Prod. 1

```
blic synchronized void put(
    ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condSignal();
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementTy
    (Elemen
  nextOut =
  size--;
  nonFull.c
  return retVal;
}
```

Signaled process enters only to leave!

Unnecessary context switch

Make Signal a Return, Let Signaled Thread E

Entry Queue

Cons. 2

nonEmpty Condition Queue

nonFull Condition Queue

Prod. 1

```
blic synchronized void put(
  ElementType element) {
  if (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++
  nonEmp
}
public synchronized ElementType get() {
  if (size == 0) {
    nonEmpty.condWait();
  }
  ElementTy
    (ElementType) buffer[nextOut];
  nextOut =
  size--;
  nonFull.condSignal()
  return retVal;
}
```

```
nonEmpty.condSignal();
get();
```

A single signal allowed

No statement allowed after signal

Make signal a hint

Hint vs. Absolute

Absolute: Application is provided information with absolute guarantee

A la non probabilistic algorithm

The server is on machine M a la the person is at address A

The condition for which you were waiting is true

Hint: Application is provided information that is likely to be true but can be checked

A la probabilistic algorithm

The server is probably on machine M, and if the message bounces, check with some central authority at some cost

The condition for which you are waiting is probably true, but must check

HINT VS. ABSOLUTE

Programmer must have code to check truth

If check fails, operations are more expensive

Usual case is efficient, unusual case is possible but not efficient

Servers do not move usually, condition does not usually get violated

Signal is a Hint

Entry Queue

Cons. 2

nonEmpty Condition Queue

nonFull Condition Queue

Urgent Queue

Cons. 1

Prod. 1

```
blic synchronized void put(
   Element
if (size
   nonFul
}
buffer[nextIn] = element;
nextIn = (nextIn + 1) % MAX_SIZE;
size++;
nonEmpty.condSignal();
}
public synchronized ElementType get() {
 f (size
   nonEmpt
}
ElementT
   (Ele
nextOu
size--;
nonFu
retur
}
```

```
while  (size >= MAX_SIZE) {
     nonFull.condWait();
}
```

```
while  (size == 0) {
     nonEmpty.condWait();
}
```

Signaled thread in urgent Q, not guaranteed state at signal time

Signaled thread must recheck condition

Make usual case efficient and other cases possible

Hint vs. Absolute Programming

```
if (!okToProceed) {
     condSynchronizer.condWait();
 }
```

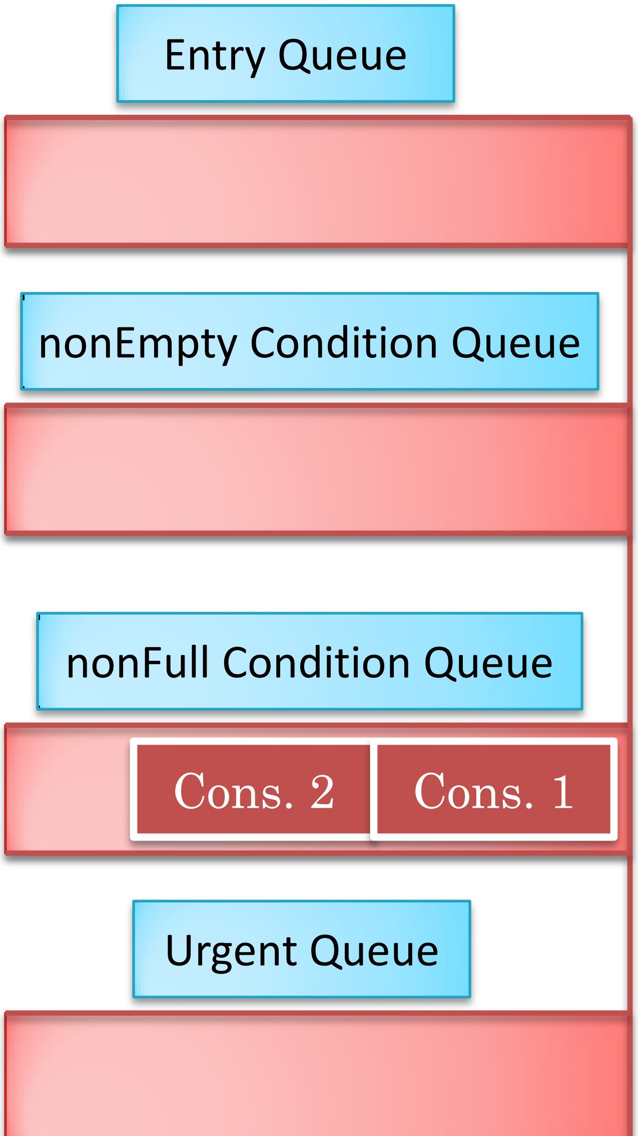
```
while (!okToProceed) {
     condSynchronizer.condWait();
 }
```

Polling?

Not polling as each iteration has a wait

Allows broadcast: all waiting threads are unblocked

BROADCAST



Prod. 1

```
blic synchronized void put(
    ElementType element) {
  while (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condBroadcastSignal();
}
public synchronized ElementType get() {
  while (size == 0) {
    nonEmpty.condWait();
   }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Broadcast

Entry Queue

nonEmpty Condition Queue

nonFull Condition Queue

Urgent Queue

Cons. 2 Cons. 1

Prod. 1

```
blic synchronized void put(
  ElementType element) {
  while (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condBroadcastSignal();
}
public synchronized ElementType get() {
  while (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
    (Ele
  nextOu
  size--
  nonFull.condSignal()
  return retVal;
}
```

Put all threads on condition Q of condition synchronizer in urgent Q

First Signaled Thread Enters

Entry Queue

nonEmpty Condition Queue

nonFull Condition Queue

Urgent Queue

Cons. 2

Cons. 1

```
blic synchronized void put(
    ElementType element) {
  while (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condBroadcastSignal();
}
public synchronized ElementType get() {
  while (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

First Signaled Thread Finishes

Entry Queue

nonEmpty Condition Queue

nonFull Condition Queue

Urgent Queue

Cons. 2

Cons. 1

```
blic synchronized void put(
    ElementType element) {
  while (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condBroadcastSignal();
}
public synchronized ElementType get() {
  while (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Second Signaled Thread Enters

Entry Queue

nonEmpty Condition Queue

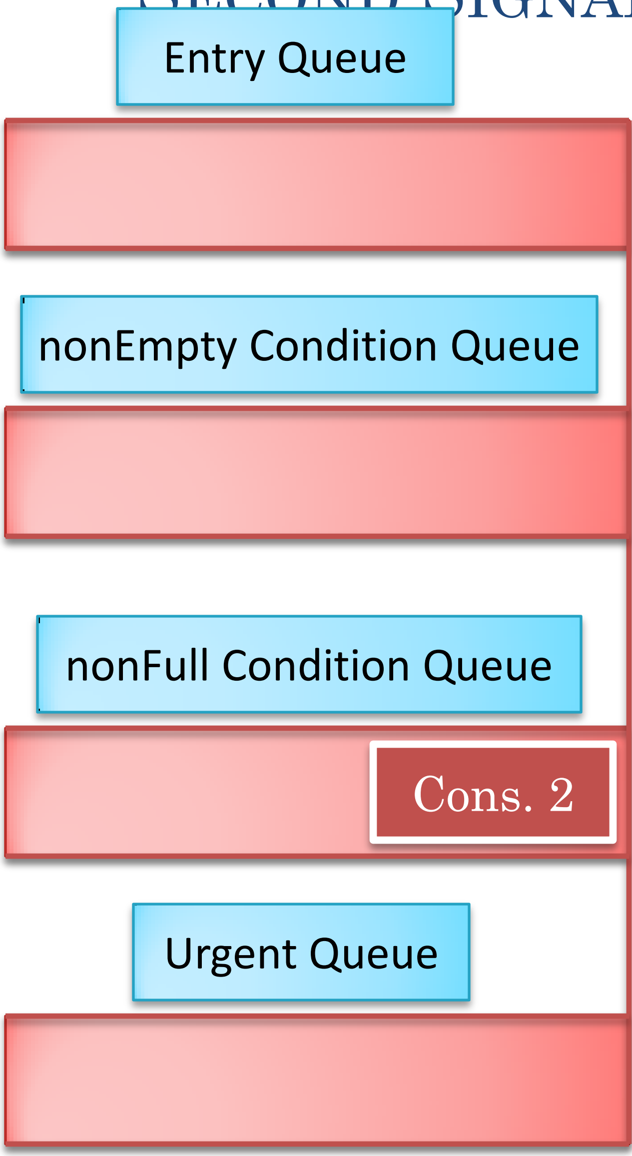
nonFull Condition Queue

Urgent Queue

Cons. 2

```
blic synchronized void put(
    ElementType element) {
  while (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condBroadcastSignal();
}
public synchronized ElementType get() {
  while (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Second Signaled Re-Enters Condition Q




```
public synchronized void put(
    ElementType element) {
  while (size >= MAX_SIZE) {
    nonFull.condWait();
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  nonEmpty.condBroadcastSignal();
}
public synchronized ElementType get() {
  while (size == 0) {
    nonEmpty.condWait();
  }
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  nonFull.condSignal()
  return retVal;
}
```

Hint Pros and Cons

Extra check for each signaled thread

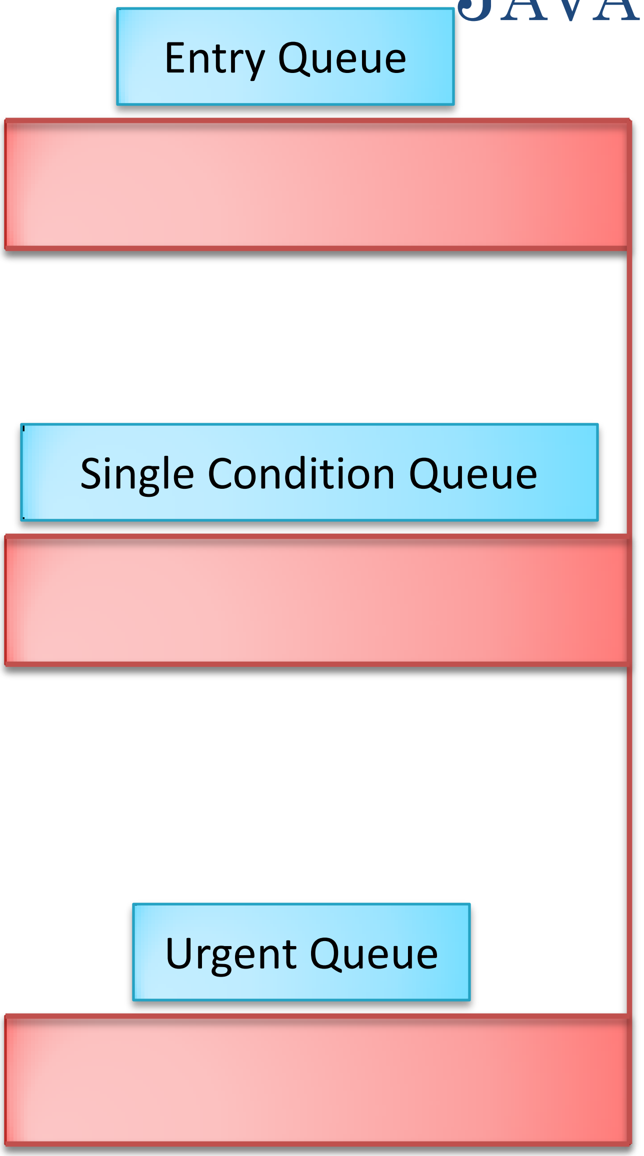
Signaled thread may in unusual cases wait again

Signaling thread does not have to restore invariant

Usual case avoids context switch (faster than extra check) or restricted signal (on return)

Allows broadcast semantics

JAVA HI

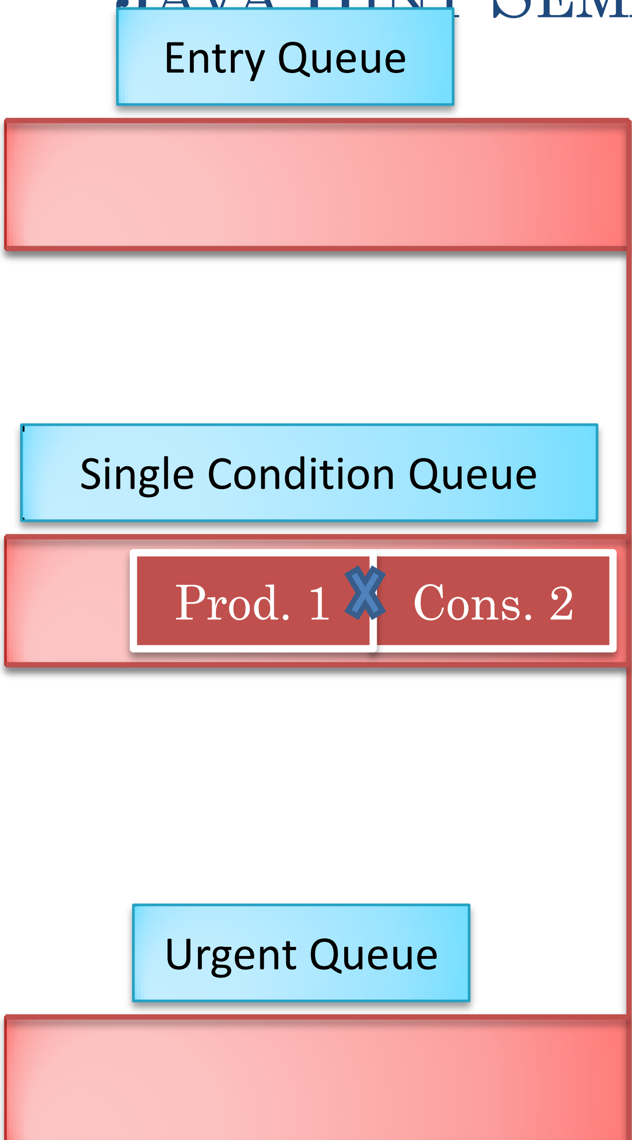



```
public synchronized void put(
    ElementType element) {
  while (size >= MAX_SIZE) {
    try{
      this.wait();
    } catch(Exception e) {}
  }
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
  size++;
  this.notify();
}
                                  ) {
    try {
      this.wait();
    } catch (Exception e) {}
  }
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  this.notify()
  return retVal;
```

Single hint-based condition synchronizer and Q in each monitor, named by object (this)

Both producer and consumer waiting on same condition synchronizer but different conditions

Java Hint Semantics for Bounded Buffer




```
public synchronized void put(
    ElementType element) {
  while (size >= MAX_SIZE) {
    try{
      this.wait();
    } catch(Exception e) {}
  }
  size++;
  this.notify();   this.notifyAll()
}
public synchronized ElementType get() {
      this.wait();
  }
  nextOut = (nextOut + 1) % MAX_SIZE;
  size--;
  this.notify()   this.notifyAll()
  return retVal;
}
```

notifyAll for Bounded Buffer?

Buffer cannot be full and empty at the same time

Bounded Buffer is driving problem

Multiple conditions share a synchronizer, hence modified name

Java vs. General Hint Semantics

Every object has an entry and condition queue

Probably why primitive values are not objects as in Smalltalk

May need broadcast semantics even when one thread eligible

Difficult to imagine cases when that happens

No need for special class labeling as monitor

No need to declare condition synchronizer

Always NonEmpty, NonFull Buffer

```
public void put(
     ElementType element) {
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;

}
public ElementType get() {
  ElementType retVal =
     (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  return retVal;
}
```

Puts cannot be executed concurrently

Gets cannot be executed concurrently

Puts and gets can be executed concurrently

Atomic Methods?

```
public synchronized void put(
    ElementType element) {
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;

}
public synchronized ElementType get() {
  ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  return retVal;
}
```

Puts and gets can be executed concurrently

Conditions?

```
public void put(
     ElementType element) {
   if (putLocked) {
      wait();
   buffer[nextIn] = element;
   nextIn = (nextIn + 1) % MAX_SIZE;
   putLocked = false;
   notifyAll();

}
public ElementType get() {
   ElementType retVal =
     (ElementType) buffer[nextOut];
   if (getLocked) {
      wait();
   getLocked = true;
   nextOut = (nextOut + 1) % MAX_SIZE;
   getLocked = false;
```

Condition synchronizer operations and associated checks should occur in atomic methods

Synchronized Statement Block


putLock


getLock

```
public void put(
      ElementType element) {
   synchronized (putLock) {
      buffer[nextIn] = element;
      nextIn = (nextIn + 1) % MAX_SIZE;
   }

}
public ElementType get() {
   ElementType retVal =
      (ElementType) buffer[nextOut];
   synchronized (getLock) {
      getLocked = true;
                                   % MAX_SIZE;
   }
```

putLock and getLock are monitors with no atomic methods associated with statement block

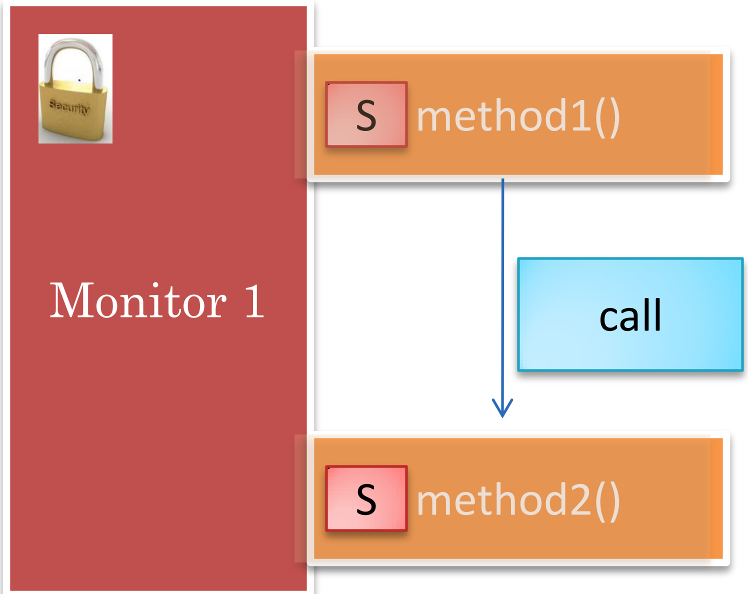
synchronize(object) { <statement list> } gets monitor lock at start of statement list and releases it on end, without entering monitor

Lock Declarations

```
public class ABoundedBuffer<ElementType>
    implements BoundedBuffer<ElementType>{
      public static final int  MAX_SIZE = 10;
      Object[] buffer = new Object[MAX_SIZE];
      int nextIn = 0;
      int nextOut = 0;
      Object putLock =  new Object();
      Object getLock =  new Object();
```

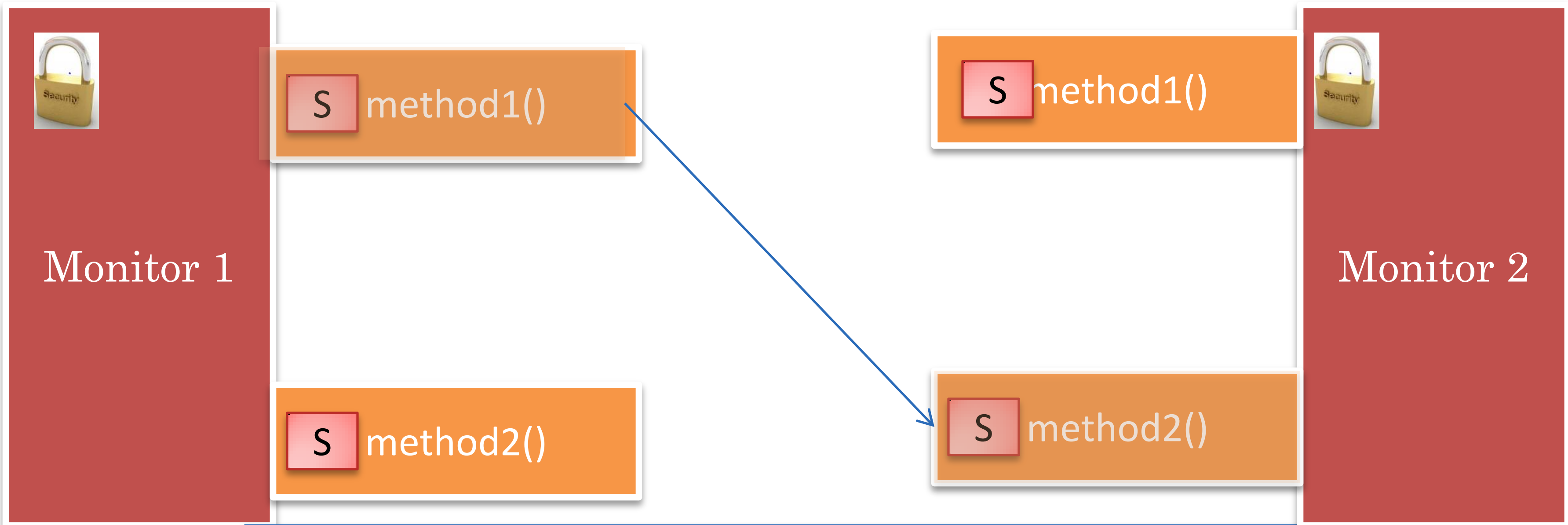
putLock and getLock are monitors with no atomic methods: Object instances

Synchronized Calling Another Synchronized Method in Same Monitor



Once a thread has the monitor lock, it can execute any number of atomic(and non atomic) methods before returning from initial method

Synchronized Calling Another Synchronized Method in Same Monitor

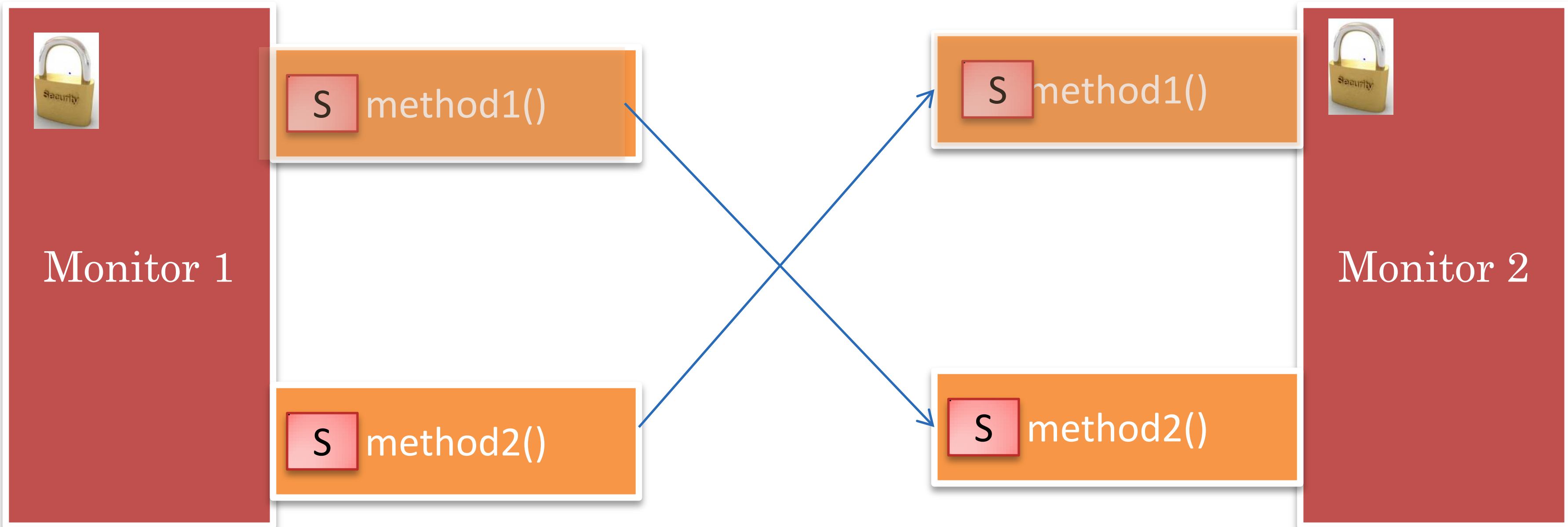


When synchronized method in one monitor calls synchronized method in another monitor, it does not release lock of first monitor

Does not have to re-get lock and restore monitor invariant before calling external method, can execute both atomically

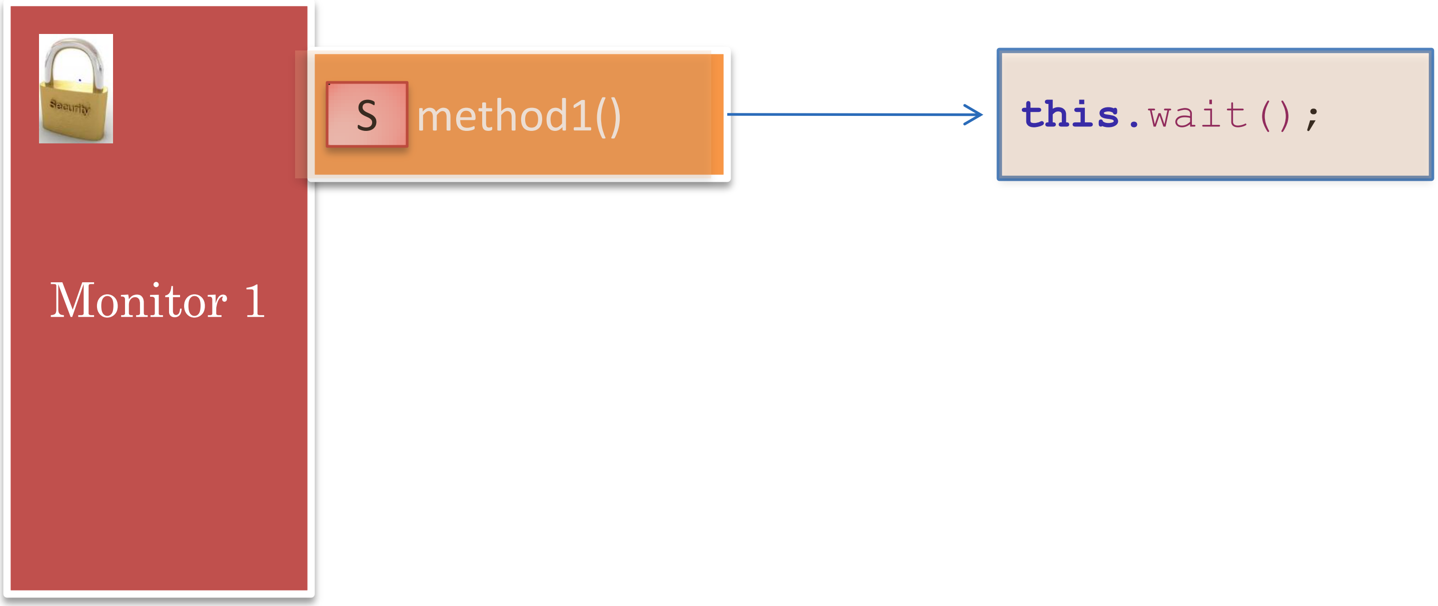
Can lead to deadlocks

Deadlock (Deadly Embrace)



Some designs release lock when calling external method

Waiting in Synchronized Method



Lock released when wait called on this in a synchronized method

Waiting in External Monitor in Synchronized Method

Monitor 1

S method1()

```
putLock.wait();
```

In Java: No-op (not illegal) when wait called on another monitor in a synchronized method

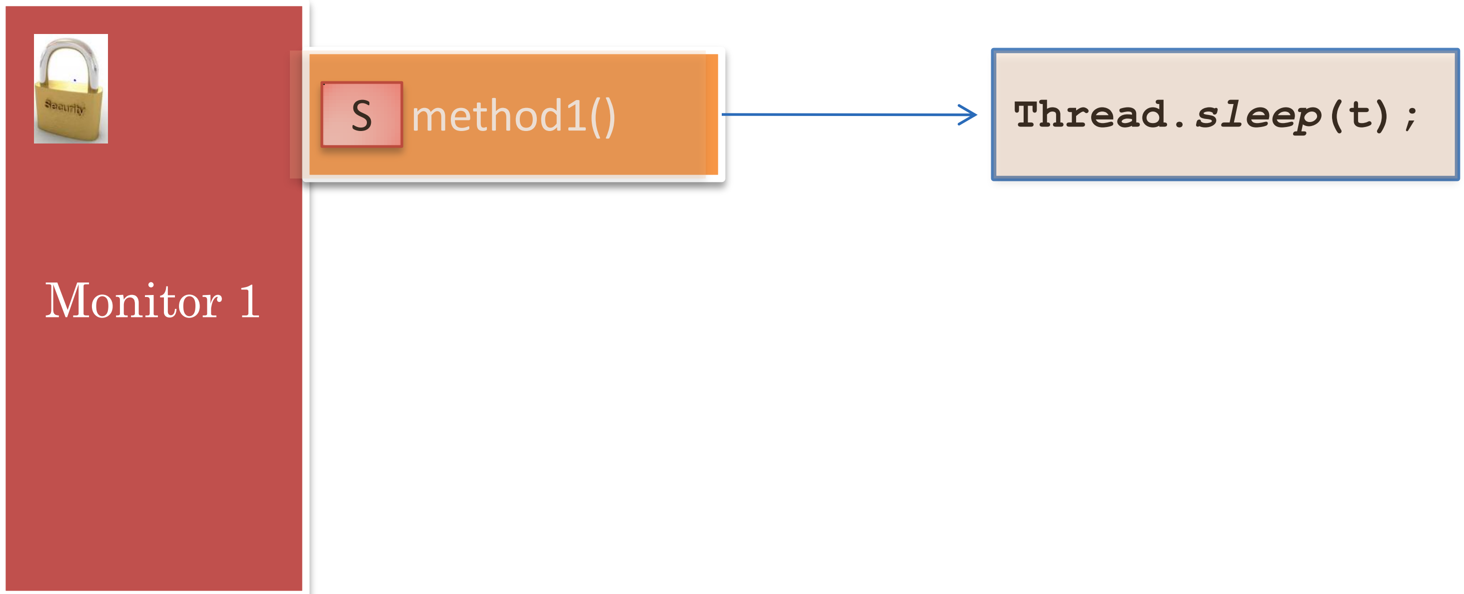
Waiting in External Monitor in Synchronized Method

```
synchronized(putLock) {
  try {
    putLock.wait();
  } catch (Exception e) {}
  …
}
```

In Java: Synchronized statement block can wait on monitor associated with block

Can wait on or notify condition queue of monitor iff you have that monitor lock

Sleeping in Synchronized Method

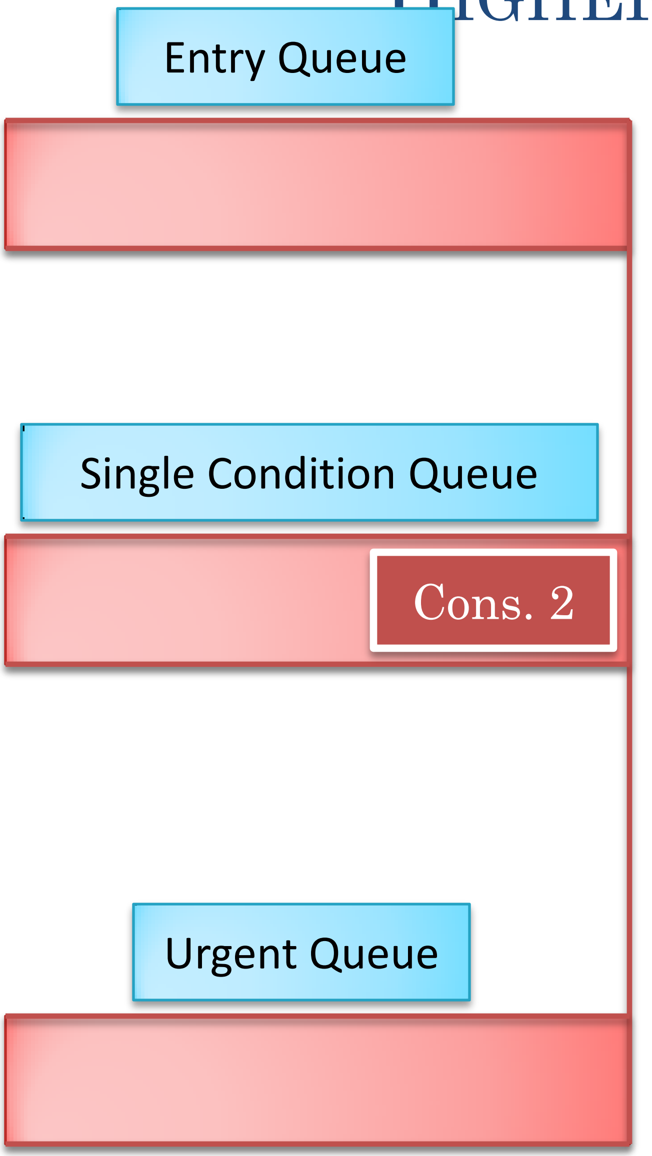


In Java: all waiting calls keep the lock

Even though waiting method can do nothing and Java knows it is waiting

Better semantics: require invariant to be restore when lock is gotten

Higher-Level Solution?




```
public synchronized void put(
    ElementType element) {
  while (size >= 
    try{
      this.wait()
    } catch(Exception e) {}
  }
  buffer[nextIn] 
  nextIn = (nextI
  size++;
  this.notify();
}
```

Declarative for mutual exclusion

Procedural for (other) synchronization

Declarative Synchronization?

Hint: FSA vs. Regular Expressions?

Vocabulary is methods

Remove All Synchronization Code

```
public synchronized void put(
     ElementType element) {
   while (size >= MAX_SIZE) {
     this.wait();
   }
   buffer[nextIn] = element;
   nextIn = (nextIn + 1) % MAX_SIZE;
   size++;
   this.notify();
}
public synchronized ElementType get() {
   while (size == 0) {
     this.wait();
   }
   ElementType retVal =
     (ElementType) buffer[nextOut];
   nextOut = (nextOut + 1) % MAX_SIZE;
   size--;
   this.notify()
   return retVal;
}
```

No synchronized keyboard

No wait and notify

No check for ok to proceed

No increment/decrement of size

Bounded Buffer (Data Structures)

```
public class ABoundedBuffer<ElementType>
     implements BoundedBuffer<ElementType>{
        public static final int  MAX_SIZE = 10;
        Object[] buffer = new Object[MAX_SIZE];
        int size = 0;
        int nextIn = 0;
        int nextOut = 0;
```

No size field

Regular vs. Path Expressions

Regular	Path
Vocabulary = $\{a_1, .. a_N\}$	Vocabulary = $\{m_1, .. m_N\}$
Usually characters	Method calls
Defines legal sequences of vocabulary elements	Defines legal sequences of method calls in a module
a | d	get, put
either a or d is acceptable	either get or put can be called

Regular vs. Path Expressions

Regular	Path
$R = a_i$	$P = m_i$
R = R1 R2	P = P1;P2
R1 must be followed by R2	P1 must be called after P2
R = R1 | R2	P = P1,P2
R1 or R2 is legal	Call to P1 or P2 can be made
R*	P = [P]
Zero or more occurrences of R can be matched next	Zero or more occurrences of call to P can be executed
	N: (P)
	Up to N concurrent executions of P can be executed

Regular vs. Path Expressions (Review)

Regular	Path
Vocabulary = $\{a_1, .. a_N\}$	Vocabulary = $\{m_1, .. m_N\}$
Usually characters	Method calls
Defines legal sequences of vocabulary elements	Defines legal sequences of method calls in a module
a | d	get, put
either a or d is acceptable	either get or put can be called

Regular vs. Path Expressions (Review)

Regular	Path
$R = a_i$	$P = m_i$
R = R1 R2	P = P1;P2
R1 must be followed by R2	P1 must be called after P2
R = R1 | R2	P = P1,P2
R1 or R2 is legal	Call to P1 or P2 can be made
R*	P = [P]
Zero or more occurrences of R can be matched next	Zero or more occurrences of call to P can be executed
	N: (P)
	Up to N concurrent executions of P can be executed

Specifying Mutual Exclusion

```
public void put(
    ElementType element) {
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
}
public ElementType get() {
ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  return retVal;
}
```

put and get can be executed concurrently, as no shared size variable

[get, put]

[get], [put]

get, put

An arbitrary number of activations of get or put and get can be executed concurrently, [] implicit on outermost expression

1: (get, put]

Only one activation of get or put can be executed concurrently

1: (get) , 1: (put)

Only one execution of get or put can be active at any one time

Specifying Synchronization

```
public void put(
    ElementType element) {
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
}
public ElementType get() {
ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  return retVal;
}
```

Must define an order between get and put

Specifying Synchronization

```
public void put(
    ElementType element) {
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
}
public ElementType get() {
ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  return retVal;
}
```

Puts can execute concurrently with another put

No mutual exclusion!

[put; get]

put; get

Each activation of get preceded by put, an infinite number of concurrent put; get sequences

1 :(put; get)

A single put; get sequence can be active at any one time

The twp methods alternate, BB of size 1

N :(put; get)

Upto N activations of (put;get) can be active at any one time

Specifying Mutual Exclusion and Synchronization

```
public void put(
    ElementType element) {
  buffer[nextIn] = element;
  nextIn = (nextIn + 1) % MAX_SIZE;
}
public ElementType get() {
ElementType retVal =
    (ElementType) buffer[nextOut];
  nextOut = (nextOut + 1) % MAX_SIZE;
  return retVal;
}
```

N : (path expression) gives global restriction, not within subexpression

Mutual Exclusion

1: (get) , 1: (put)

Only one execution of get or put can active

Synchronization

N :(put; get)

N activations of (put;get) can be active at any one time

Both

N :(1:(put); 1:(get))

N activations of (put;get) can be active but only one activation of put and one activation of get

Reader Writer Problem

There can be an arbitrary number of readers or a single writer active at any one time

1: ([read], write)

Fair readers writers: writer does not wait for a reader who comes after it

Path expression cannot handle all synchronization schemes

Regular expression cannot handle all languages

Path Expressions

Method calls are matched according to specified path expressions

Method calls that do not match are blocked

After top-level path expression is matched , it can be matched again

1: (get, put)

Implementation unspecified: when is unblocking checking checked and which waiting method unblocked

Not practical but helps understand synchronization better