

Monitors

CS511

Review

- ▶ We've seen that **semaphores** are an efficient tool to solve synchronization problems
- ▶ However, they have some drawbacks
 1. They are low-level constructs
 - ▶ It is easy to forget an acquire or release
 2. They are not related to the data
 - ▶ They can appear in any part of the code

Monitors

- ▶ Combines ADTs and mutual exclusion
 - ▶ Proposed by Tony Hoare:
Monitors: An Operating System Structuring Concept (Communications of the ACM, 17(10), 549-557, 1974).
- ▶ Adopted in many modern PLs
 - ▶ Java
 - ▶ C#
 - ▶ Python
 - ▶ Ruby

Main Ingredients

- ▶ A set of operations encapsulated in modules
- ▶ A unique **lock** that ensures mutual exclusion to all operations in the monitor
- ▶ Special variables called **condition variables**, that are used to program conditional synchronization

Counter Example

- ▶ Construct a counter with two operations:
 - ▶ `inc()`
 - ▶ `dec()`
- ▶ No two threads should be able to simultaneously modify the value of the counter
 - ▶ Think of a solution using semaphores
 - ▶ A solution using monitors

Counter using Semaphores

```
1 class Counter {
2     private int c = 0;
3     private Semaphore mutex = new Semaphore(1);
4
5     public void inc() {
6         mutex.acquire();
7         c++;
8         mutex.release();
9     }
10    public void dec() {
11        mutex.acquire();
12        c--;
13        mutex.release();
14    }
15 }
```

Counter as Monitor in Groovy (and Java)

```
1 class Counter {  
2  
3     private int counter = 0;  
4  
5     public synchronized void inc() {  
6         counter++;  
7     }  
8  
9     public synchronized void dec() {  
10        counter--;  
11    }  
12  
13 }
```

- ▶ Each object has its own lock called **intrinsic** or **monitor** lock
- ▶ It also has its own **wait-set** (more on this later)

Counter as Monitor in Groovy – Continued

```
1 Counter c = new Counter();
2
3 def P = Thread.start {
4     10.times {
5         c.inc();
6     }
7 }
8
9 def Q = Thread.start {
10     10.times {
11         c.inc();
12     }
13 }
14
15 P.join();
16 Q.join();
17 println c.counter;
```


Counter Monitor in Pseudocode

```
1 monitor Counter {  
2     private int counter = 0;  
3  
4     public void inc() {  
5         counter++;  
6     }  
7  
8     public void dec() {  
9         counter--;  
10    }  
11  
12 }
```

- Note the use of the keyword `monitor` and the missing qualifier `synchronized`

Condition Variables

- ▶ Apart from the lock, there are **condition variables** associated to the monitor
- ▶ They have
 1. Three operations:
 - ▶ `Cond.wait()/await()`
 - ▶ `Cond.notify()/signal()`
 - ▶ `Cond.notifyAll()/signalAll()`
 2. A set of blocked processes.

Condition Variables

`Cond.wait()/await()`

- ▶ **Always** blocks the process and places it in the waiting set of the variable `Cond`.
- ▶ When it blocks, it releases the mutex on the monitor.

`Cond.notify()/signal()`

- ▶ Unblocks the first process in the waiting set of the variable `Cond` and sets it to the **READY** state
- ▶ If there are no processes in the waiting set, it has no effect.

`Cond.notifyAll()/signalAll()`

- ▶ Unblocks all the processes in the waiting set of the variable `Cond` and sets them to the **READY** state
- ▶ If there are no processes in the waiting set, it has no effect.

Example: Buffer of Size 1 in Groovy

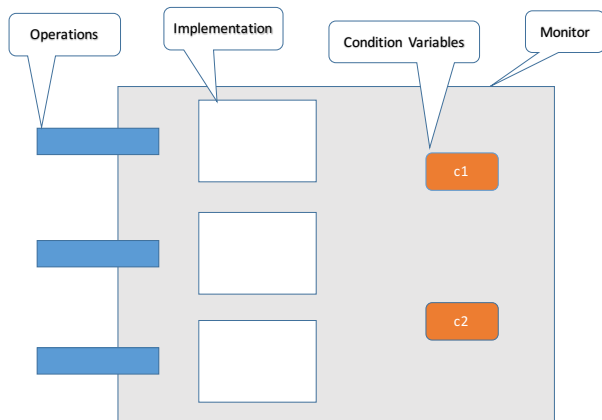
```
1 class Buffer {
2     Object buffer = null; // shared buffer
3
4     synchronized Object consume() {
5         while (buffer == null)
6             wait(); // wait on object's wait-set
7         Object aux = buffer;
8         buffer = null;
9         notifyAll(); // signal on object's wait-set
10        return aux;
11    }
12
13    synchronized void produce(Object o) {
14        while (buffer != null)
15            wait(); // wait on object's wait-set
16        buffer = o;
17        notifyAll(); // signal on object's wait-set
18    }
19 }
```

- ▶ Every object has a wait-set on which methods `wait`, `notify` and `notifyAll` operate
- ▶ These methods must be called from synchronized methods or else an `IllegalMonitorStateException` is raised

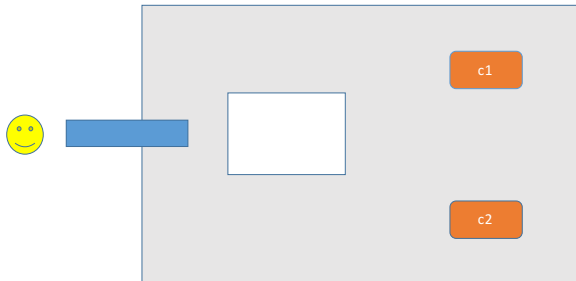
Example: Buffer of Size 1 in Groovy (cont)

```
1 Buffer b =new Buffer();
2
3 20.times {
4     int id=it
5     Thread.start {
6         println "consumer "+ b.consume();
7     }
8 }
9
10 20.times {
11     int id=it
12     Thread.start {
13         b.produce(id);
14     }
15 }
16
17 return ;
```

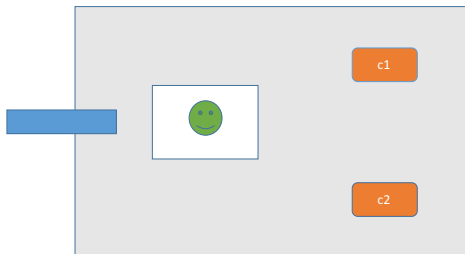
Explaining Monitors Graphically



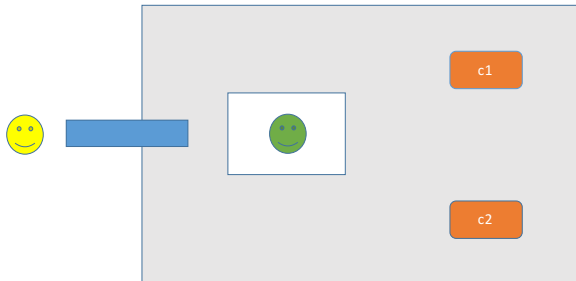
Typical Behavior



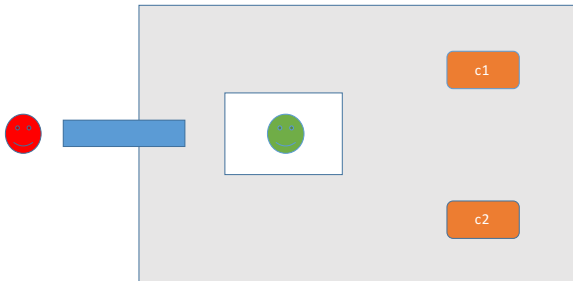
Typical Behavior



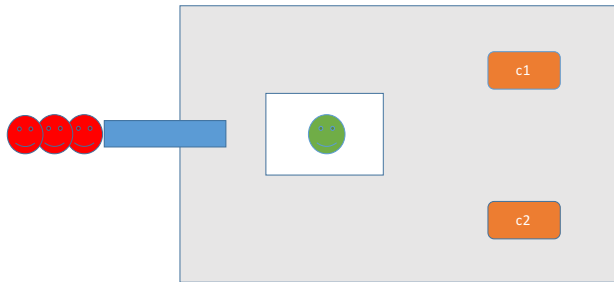
Typical Behavior



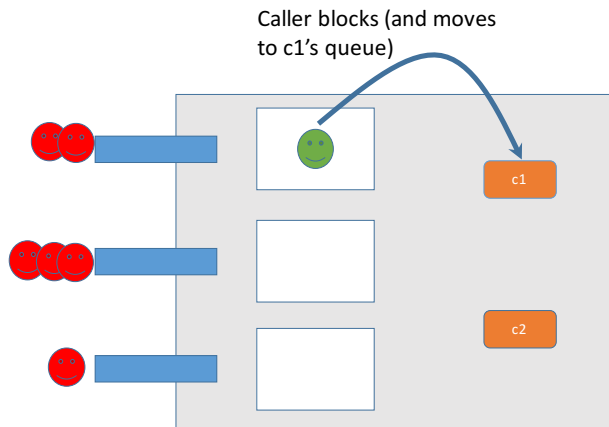
Typical Behavior



Typical Behavior

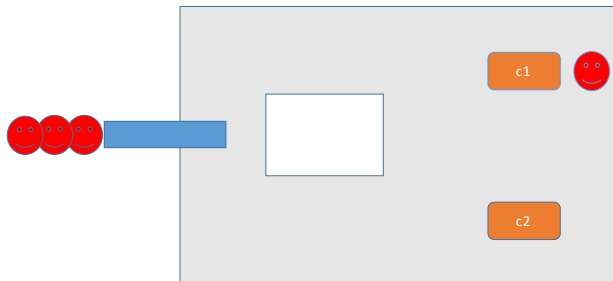


Wait



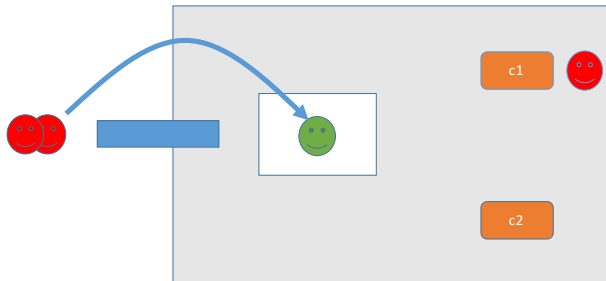
- ▶ Blocks process currently executing and associates it to variable's waiting set
- ▶ Upon blocking frees the **lock** allowing the entry of other processes

Wait



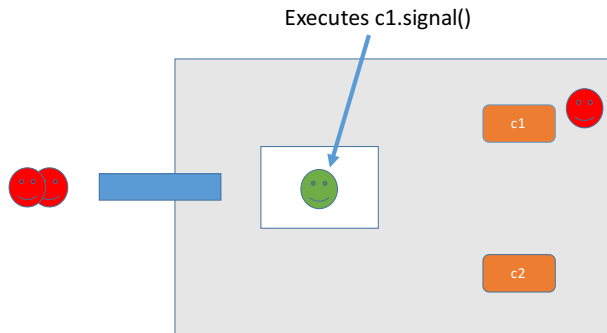
- ▶ Blocks process currently executing and associates it to variable's waiting set
- ▶ Upon blocking frees the **lock** allowing the entry of other processes

Wait



- ▶ Blocks process currently executing and associates it to variable's waiting set
- ▶ Upon blocking frees the **lock** allowing the entry of other processes

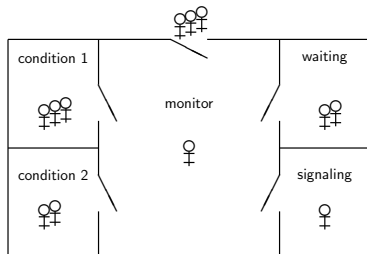
Signal



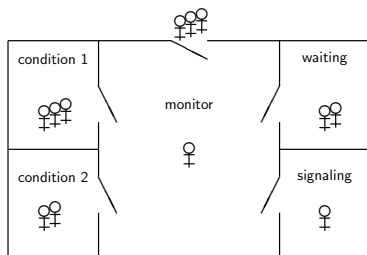
- ▶ Signalling process continues to execute after notifying on `c1`?
- ▶ Processes waiting in `c1`'s waiting-set start immediately running inside the monitor?
- ▶ What about the processes blocked on entry to the monitor?

Signal – States That a Process Can Be In

- ▶ Waiting to enter the monitor
- ▶ Executing within the monitor (only one)
- ▶ Blocked on condition variables
- ▶ Set of processes just released from waiting on a condition variable
- ▶ Set of processes that have just completed a `signal` operation



Notify



Two strategies:

- ▶ **Signal and Urgent Wait:** $E < S < W$ (classical monitors)
- ▶ **Signal and Continue:** $E = W < S$ (Java \Leftarrow We focus on this one)

where the letters denote the precedence of

- ▶ S : signalling processes
- ▶ W : waiting processes
- ▶ E : processes blocked on entry

Monitors

More Examples

Condition Variables

Monitors in Java

Visibility

Explicit Locks

Buffer of Size 1 in Groovy - Discussion

```
1 class Buffer {
2     Object buffer = null; // shared buffer
3
4     synchronized Object consume() {
5         while (buffer == null)
6             wait(); // wait on object's wait-set
7         Object aux = buffer;
8         buffer = null;
9         notifyAll(); // signal on object's wait-set
10        return aux;
11    }
12
13    synchronized void produce(Object o) {
14        while (buffer != null)
15            wait(); // wait on object's wait-set
16        buffer = o;
17        notifyAll(); // signal on object's wait-set
18    }
19 }
```

Buffer of Size 1 in Groovy - Discussion

```
1 class Buffer {
2     Object buffer = null; // shared buffer
3
4     synchronized Object consume() {
5         while (buffer == null)
6             wait(); // wait on object's wait-set
7         Object aux = buffer;
8         buffer = null;
9         notify(); // signal on object's wait-set
10        return aux;
11    }
12
13    synchronized void produce(Object o) {
14        while (buffer != null)
15            wait(); // wait on object's wait-set
16        buffer = o;
17        notify(); // signal on object's wait-set
18    }
19 }
```

► What goes wrong and why?

Monitor that Defines a Semaphore

```
1 class Semaphore {
2
3     private int permissions;
4
5     Semaphore(int n) {
6         this.permissions = n;
7     }
8
9     synchronized void acquire() {
10         while (permissions == 0)
11             wait();
12         permissions--;
13     }
14
15     synchronized void release() {
16         permissions++;
17         notifyAll();
18     }
19
20 }
```

Monitor that Defines a Semaphore

```
1  c=0;
2  Semaphore mutex = new Semaphore(1);
3  def P =
4      Thread.start {
5          20.times{
6              mutex.acquire();
7              c++;
8              mutex.release();
9          }
10     }
11
12 def Q =
13     Thread.start {
14         20.times {
15             mutex.acquire();
16             c++;
17             mutex.release();
18         }
19     }
20
21 P.join();
22 Q.join();
23 println c;
```

Signal and Continue

- ▶ Must re-check the condition since may have gained entry long after it was notified

```
synchronized void acquire() {  
    while (permissions == 0)  
        nonZero.wait();  
    permissions--;  
}
```

Another reason for re-checking the condition:

- ▶ Spurious wakeups: “Implementations are permitted, although not encouraged, to perform “spurious wake-ups”, that is, to remove threads from wait sets and thus enable resumption without explicit instructions to do so.”¹

¹JLS for Java SE 17 (page 740).

Signal and Continue

```
1 class Semaphore {
2     private int permissions;
3
4     Semaphore(int n) {
5         this.permissions = n;
6     }
7
8     synchronized void acquire() {
9         while (permissions == 0)
10             wait();
11         permissions--;
12     }
13
14     synchronized void release() {
15         permissions++;
16         notifyAll();
17     }
18
19 }
```

► Is it fair?

► See *Specific Notification for Java Thread Synchronization* by

Tom Cargill, 1996.²

²<http://www.dre.vanderbilt.edu/%7Eschmidt/PDF/specific-notification.pdf>

Monitors

More Examples

Condition Variables

not explain in class

Monitors in Java

Visibility

Explicit Locks

Buffer of Size 1 in Groovy - Discussion

```
1 class Buffer {
2     Object buffer = null; // shared buffer
3
4     synchronized Object consume() {
5         while (buffer == null)
6             wait(); // wait on object's wait-set
7         Object aux = buffer;
8         buffer = null;
9         notifyAll(); // signal on object's wait-set
10        return aux;
11    }
12
13    synchronized void produce(Object o) {
14        while (buffer != null)
15            wait(); // wait on object's wait-set
16        buffer = o;
17        notifyAll(); // signal on object's wait-set
18    }
19 }
```

- ▶ Inefficient
- ▶ Much more efficient (and clearer) to have multiple wait-sets

Condition Variables

```
1 import java.util.concurrent.locks.*;
2
3 class Buffer {
4     Object buffer = null; // shared buffer
5     final Lock lock = new ReentrantLock();
6     final Condition empty = lock.newCondition();
7     final Condition full = lock.newCondition();
8
9     Object consume() {
10         lock.lock();
11         try {
12             while (buffer == null)
13                 full.await();
14             Object aux = buffer;
15             buffer = null;
16             empty.signal();
17             return aux;
18         } finally {
19             lock.unlock();
20         }
21     }
22
23     // continues in next slide
24 }
```

Condition Variables

```
1    void produce(Object o) {  
2        lock.lock();  
3        try {  
4            while (buffer != null)  
5                empty.await();  
6            buffer = o;  
7            full.signal();  
8        } finally {  
9            lock.unlock();  
10       }  
11    }  
12 }
```

Condition Variables

```
1 Buffer b =new Buffer();
2
3
4 20.times {
5     int id=it
6     Thread.start {
7         println "consumer "+ b.consume();
8     }
9 }
10
11
12 20.times {
13     int id=it
14     Thread.start {
15         b.produce(id);
16     }
17 }
18
19
20
21 return ;
```

Buffer of Size n

```
1  monitor Buffer {
2      private Object[] data = new Object[N];
3      private int begin = 0, end = 0;
4      private condition notFull, notEmpty;
5
6      public void put(Object o) {
7          while (isFull())
8              notFull.wait();
9          data[begin] = o;
10         begin = (begin+1) % N;
11         notEmpty.notify()
12     }
13
14     public Object take() {
15         while (isEmpty())
16             notEmpty.wait();
17         Object result = data[end];
18         end = (end+1) % N
19         notFull.notify();
20         return result;
21     }
22
23     private boolean isEmpty() { return begin == end; }
24     private boolean isFull()  { return next(begin) == end; }
25 }
```

Readers/Writers

```
1 monitor RW {  
2     ...  
3  
4     public void read() {  
5         ...  
6     }  
7  
8     public void write() {  
9         ...  
10    }  
11  
12 }
```

What is the problem with this setting?

Readers/Writers

```
1 monitor RW {
2     int readers = 0;
3     int writers = 0;
4     condition OKtoRead, OKtoWrite;
5
6     public void StartRead() {
7         while (writers != 0 or not OKtoWrite.empty()) {
8             OKtoRead.wait();
9         }
10        readers = readers + 1;
11    }
12
13    public void EndRead {
14        readers = readers - 1;
15        if (readers==0) {
16            OKtoWrite.notify();
17        }
18    }
19    // continues
```


Readers/Writers

```
1  public void StartWrite() {
2      while (writers != 0 or readers != 0) {
3          OKtoWrite.wait();
4      }
5      writers = writers + 1;
6  }
7
8  public void EndWrite() {
9      writers = writers - 1;
10     OKtoWrite.signal();
11     OKtoRead.signalAll();
12 }
13 }
```

Assessment

- ▶ Upholds the readers-writers invariant.
- ▶ However, it gives priority to readers over writers:
 - ▶ new readers can enter the monitor without waiting as long as a reader is active
 - ▶ waiting writers have to wait until the last reader calls `endRead` and signals `OKtoWrite`
 - ▶ as long as readers keep arriving and waiting to enter the monitor, the waiting writers will never execute

Dining Philosophers

```
1 monitor ForkMonitor {
2     int[] fork = {2,2,2,2,2};
3     condition[] OKtoEat; // 0-4
4
5     public void takeForks(integer i) {
6         while (fork[i] != 2) {
7             OKtoEat[i].wait();
8         }
9         fork[i+1] = fork[i+1] - 1;
10        fork[i-1] = fork[i-1] - 1;
11    }
12
13    public void releaseForks(integer i) {
14        fork[i+1] = fork[i+1] + 1;
15        fork[i-1] = fork[i-1] + 1;
16        if (fork[i+1] == 2) {
17            OKtoEat[i+1].signal();
18        }
19        if (fork[i-1] == 2) {
20            OKtoEat[i-1].signal();
21        }
22    }
23 }
```

`forks[i]` is number of forks available to philosopher *i*

Monitors

More Examples

Condition Variables

Monitors in Java

Visibility

Explicit Locks

Visibility

- ▶ Whether a thread can see the modifications of other threads
- ▶ Visibility is subtle because the compiler may
 - ▶ Reorder operations
 - ▶ Cache values in registers
- ▶ `synchronization`, used for atomicity, helps with visibility too:
 - ▶ All changes made in one synchronized method or block are visible with respect to other synchronized methods and blocks employing the same lock.

Volatile Variables

```
1 int n=0;

2 Thread.start { // P          2 Thread.start { // Q
3   int local1, local2;        3   int local;
4   n = some expression;      4   local = n+6;
5   computation not using n;  5
6   local1 = (n+5)*7;          6
7   local2 = n+5;              7
8   n = local1 * local2;       8
9 }                             9 }
```

The instruction in Q may be interleaved at any place during the execution of the instructions in P

Volatile Variables

An optimizing compiler could translate statements in thread P as:

| | |
|------------------------------|----------------------------|
| 1 tempReg1 = some expression | 1 n = some expression; |
| 2 computation not using n | 2 computation not using n; |
| 3 tempReg2 = tempReg1 + 5 | 3 local1 = (n+5)*7; |
| 4 local2 = tempReg2 | 4 local2 = n+5; |
| 5 local1 = tempReg2 * 7 | 5 n = local1 * local2; |
| 6 n = local1 * local2 | 6 . |

- ▶ No assignment to n in the first statement. Original statements p3 and p4 are executed out of order
- ▶ Without concurrency, the translated code would be correct
- ▶ With concurrency and interleaving, the translated code may no longer be correct
- ▶ Specifying a variable as volatile instructs the compiler to load and store the value of the variable at each use, rather than to optimize away these loads and stores.

Example 1³

```
1 public class SharedVariable {
2     private static int sharedVariable = 0;
3     public static void main(String[] args) throws InterruptedException
4
5     new Thread(new Runnable() {
6         @Override
7         public void run() {
8             try { Thread.sleep(100); }
9             catch (InterruptedException e) {
10                 e.printStackTrace();
11             }
12             sharedVariable = 1;
13         }).start();
14
15     for(int i=0;i<1000;i++) {
16         for(;;) {
17             if(sharedVariable == 1) { break; }
18         }
19     }
20     System.out.println("SharedVariable : " + sharedVariable);
21 }
22 }
```

Try this code as is (loops due to compiler optimization), then add the qualified volatile to the declaration of `sharedVariable` and run

Example 2

```
1 public class NoVisibility {
2     private static boolean ready;
3     private static int number;
4
5     private static class ReaderThread extends Thread {
6         public void run() {
7             while (!ready)
8                 Thread.yield();
9             System.out.println(number);
10        }
11    }
12
13    public static void main(String[] args) {
14        new ReaderThread().start();
15        number = 42;
16        ready = true;
17    }
18 }
```

- ▶ `java.lang.Thread.yield()` causes the currently executing thread object to temporarily pause and allow other threads to execute
- ▶ What is the output?

Example 2

```
1 public class NoVisibility {
2     private static boolean ready;
3     private static int number;
4
5     private static class ReaderThread extends Thread {
6         public void run() {
7             while (!ready)
8                 Thread.yield();
9             System.out.println(number);
10        }
11    }
12
13    public static void main(String[] args) {
14        new ReaderThread().start();
15        number = 42;
16        ready = true;
17    }
18 }
```

- ▶ `java.lang.Thread.yield()` causes the currently executing thread object to temporarily pause and allow other threads to execute
- ▶ What is the output? Could loop forever or print 0!

Monitors

More Examples

Condition Variables

Monitors in Java

Visibility

Explicit Locks

Explicit Locks

- ▶ Apart from the intrinsic lock of an object, one can use **explicit locks**
- ▶ This is convenient for modeling condition variables
- ▶ We next present the `Lock` interface and the class `ReentrantLock` that implements it

Explicit Locks – An Example

- ▶ We take a look at the producers/consumers example
- ▶ We present two implementations:
 - ▶ Using intrinsic locks
 - ▶ Using explicit locks
- ▶ Source: Goetz's *Java Concurrency in Practice*, Addison-Wesley, 2006

Bounded Buffers Revisited

```
1 public abstract class BaseBoundedBuffer <V> {
2     private final V[] buf;
3     private int tail;
4     private int head;
5     private int count;
6
7     protected BaseBoundedBuffer(int capacity) {
8         this.buf = (V[]) new Object[capacity];
9     }
10
11     protected synchronized final void doPut(V v) {
12         buf[tail] = v;
13         if (++tail == buf.length)
14             tail = 0;
15         ++count;
16     }
17
18     // continued
```

Bounded Buffers Revisited

```
1
2     protected synchronized final V doTake() {
3         V v = buf[head];
4         buf[head] = null;
5         if (++head == buf.length)
6             head = 0;
7         --count;
8         return v;
9     }
10
11     public synchronized final boolean isFull() {
12         return count == buf.length;
13     }
14
15     public synchronized final boolean isEmpty() {
16         return count == 0;
17     }
18 }
```

Crude Blocking

```
1 public class BoundedBuffer <V> extends BaseBoundedBuffer<V> {
2     // CONDITION PREDICATE: not-full (!isFull())
3     // CONDITION PREDICATE: not-empty (!isEmpty())
4     public BoundedBuffer() {
5         this(100);
6     }
7
8     public BoundedBuffer(int size) {
9         super(size);
10    }
11
12    // BLOCKS-UNTIL: not-full
13    public synchronized void put(V v) throws InterruptedException
14        while (isFull())
15            wait();
16        doPut(v);
17        notifyAll();
18    }
19    // continues
```


Crude Blocking

```
1    // BLOCKS-UNTIL: not-empty
2    public synchronized V take() throws InterruptedException {
3        while (isEmpty())
4            wait();
5        V v = doTake();
6        notifyAll();
7        return v;
8    }
9
10   // BLOCKS-UNTIL: not-full
11   // Alternate form of put() using conditional notification
12   public synchronized void alternatePut(V v) throws InterruptedException {
13       while (isFull())
14           wait();
15       boolean wasEmpty = isEmpty();
16       doPut(v);
17       if (wasEmpty)
18           notifyAll();
19   }
20 }
```

Lock Interface

```
1 public interface Lock {
2     void lock();                // Acquires the lock.
3     void lockInterruptibly() throws InterruptedException;
4                               // Acquires the lock unless
5                               // the current thread is interrupted.
6     boolean tryLock();          // Acquires the lock only if it is
7                               // free at the time of invocation.
8     boolean tryLock(long time, TimeUnit unit)
9         throws InterruptedException; // Acquires the lock if it
10        // is free within the given waiting time
11        // and the current thread has not been interrupted.
12     void unlock();              // Releases the lock.
13     Condition newCondition();   // Returns a new Condition instance
14                               // that is bound to this Lock instance.
15 }
```

Using Locks

```
1 Lock l = new ReentrantLock();
2 l.lock();
3 try {
4     // access the resource protected by this lock
5 } finally {
6     l.unlock();
7 }
```

Using Condition Interface

```
1 public interface Condition {
2     void await(); // Causes the current thread to wait until
3                   // it is signalled or interrupted.
4     boolean await(long time, TimeUnit unit);
5     long awaitNanos(long nanosTimeout);
6     void awaitUninterruptibly();
7     boolean awaitUntil(Date deadline);
8
9     void signal(); // Wakes up one waiting thread.
10    void signalAll(); // Wakes up all waiting threads.
11 }
```

- ▶ In condition objects we replace wait, notify and notifyAll by await, signal and signalAll

Using Condition Interface

```
1 public class ConditionBoundedBuffer <T> {
2     protected final Lock lock = new ReentrantLock();
3     private final Condition notFull = lock.newCondition();
4     private final Condition notEmpty = lock.newCondition();
5     private static final int BUFFER_SIZE = 100;
6     private final T[] items = (T[]) new Object[BUFFER_SIZE];
7     private int tail, head, count;
8
9     // BLOCKS-UNTIL: notFull
10    public void put(T x) throws InterruptedException {
11        lock.lock();
12        try {
13            while (count == items.length)
14                notFull.await();
15            items[tail] = x;
16            if (++tail == items.length)
17                tail = 0;
18            ++count;
19            notEmpty.signal();
20        } finally {
21            lock.unlock();
22        }
23    }
24    // continues
```

Using Condition Interface

```
1    // BLOCKS-UNTIL: notEmpty
2    public T take() throws InterruptedException {
3        lock.lock();
4        try {
5            while (count == 0)
6                notEmpty.await();
7            T x = items[head];
8            items[head] = null;
9            if (++head == items.length)
10                head = 0;
11            --count;
12            notFull.signal();
13            return x;
14        } finally {
15            lock.unlock();
16        }
17    }
18 }
```