

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 [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 using Monitors

```
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 }
```

- ▶ The monitor comes equipped with a **lock or mutex** that allows at most one thread to execute its operations
- ▶ Note:
 - ▶ This is pseudocode (not Hydra nor Java)

Counter in 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
- ▶ The class also has a lock but we don't use it in this example

Condition Variables

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

Condition Variables

`Cond.wait()`

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

`Cond.signal()`

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

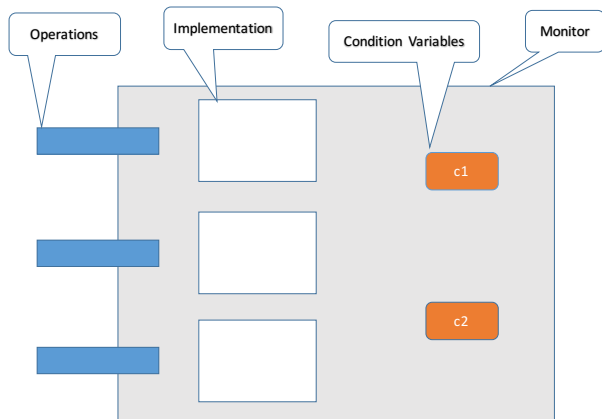
`Cond.empty()`

- ▶ Checks if waiting queue of `Cond` is empty or not

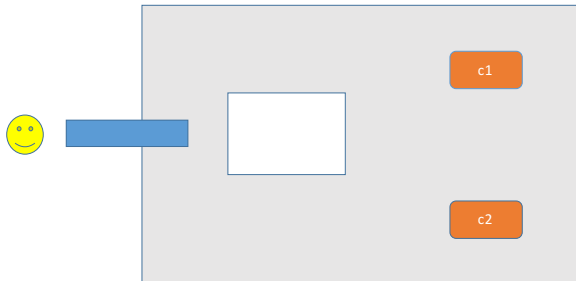
Example: Buffer of Size 1

```
1 monitor Buffer {
2     private condition full; // wait until space available
3     private condition empty; // wait until data available
4
5     private Object data = null; // shared data
6
7     public Object read() {
8         if (data == null)
9             full.wait();
10        aux = data;
11        data = null;
12        empty.signal();
13        return aux;
14    }
15
16    public void write(Object o) {
17        if (data != null)
18            empty.wait();
19        data = o;
20        full.signal();
21    }
22 }
23 }
```

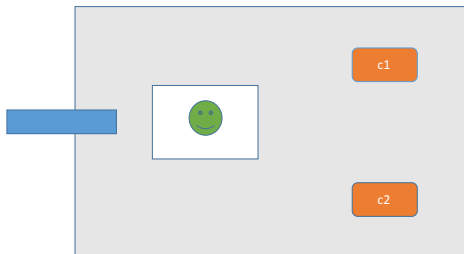
Explaining Monitors Graphically



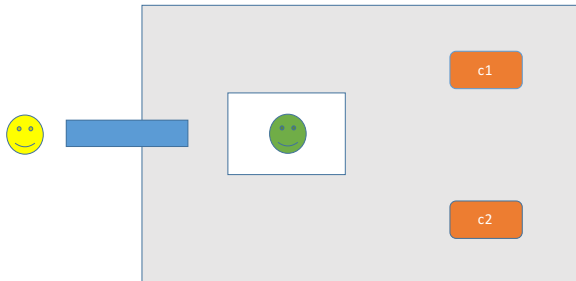
Typical Behavior



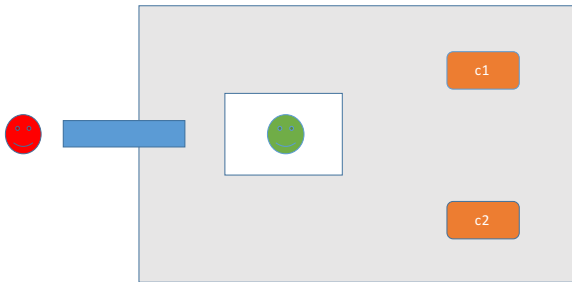
Typical Behavior



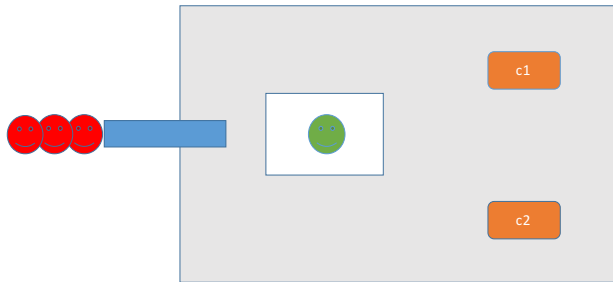
Typical Behavior



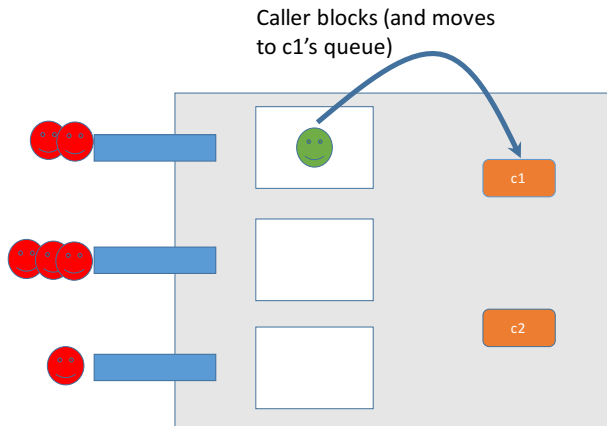
Typical Behavior



Typical Behavior

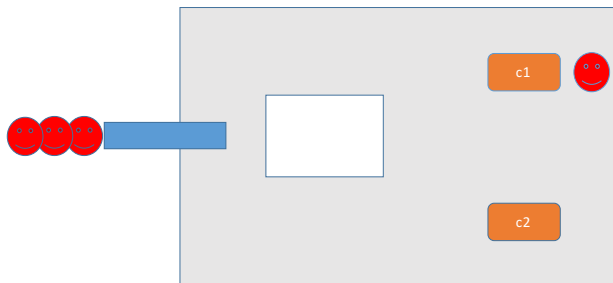


Wait



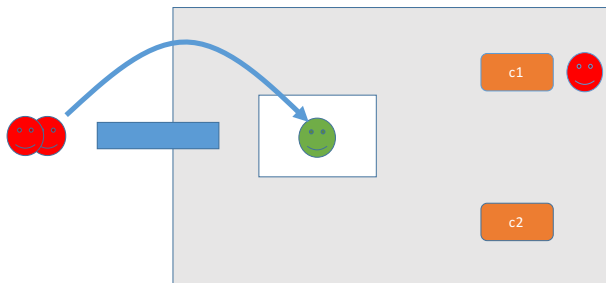
- ▶ Blocks process currently executing and associates it to variable's queue
- ▶ Upon blocking frees the **lock** allowing the entry of other processes

Wait



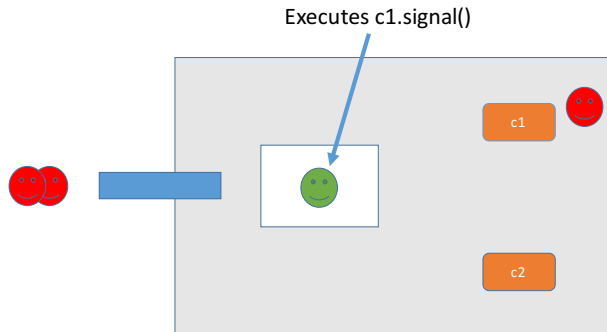
- ▶ Blocks process currently executing and associates it to variable's queue
- ▶ Upon blocking frees the **lock** allowing the entry of other processes

Wait



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

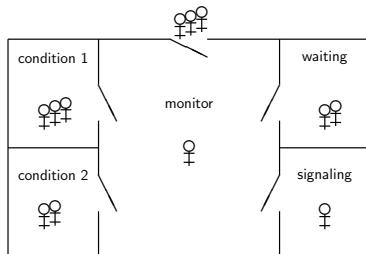
Signal



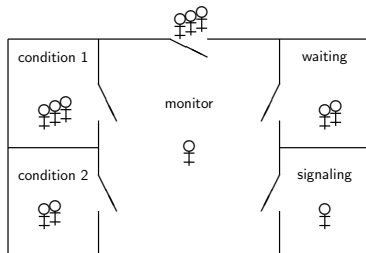
- ▶ Signalling process continues to execute after signalling on `c1`?
- ▶ Processes waiting in `c1`'s queue start immediately running inside the monitor?
- ▶ What about the processes blocked on entry to the monitor?

Signal – States That a Process Can Be In

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



Signal



- ▶ Two strategies:
 - ▶ **Signal and Urgent Wait:** $E < S < W$ (classical monitors)
 - ▶ **Signal and Continue:** $E = W < S$ (Java)
- ▶ where the letters denote the precedence of
 - ▶ S : signalling processes
 - ▶ W : waiting processes
 - ▶ E : processes blocked on entry

Signal and Urgent Wait (or Immediate Resumption Requirement – IRR)

$$E < S < W$$

- ▶ When a process blocked on a condition variable is signaled, it immediately begins executing **ahead** of the signaling process
 - ▶ Resumes at the instruction immediately following the call to **wait** that blocked it
 - ▶ No need not check the condition
- ▶ Rationale: Signaling process changed the state of the monitor so that the condition now holds
- ▶ Cons: signaling process unnecessarily delayed (unless signal is the last operation)

Example of Buffer of Size 1 Revisited

```
1 monitor Buffer {
2     private condition notFull; // wait until space available
3     private condition notEmpty; // wait until data available
4
5     private Object data = null; // shared data
6
7     public Object read() {
8         if (data == null)
9             notEmpty.wait();
10        aux = data;
11        data = null;
12        notFull.signal();
13        return aux;
14    }
15
16    public void write(Object o) {
17        if (data != null)
18            notFull.wait();
19        data = o;
20        notEmpty.signal();
21    }
22
23 }
```

Monitors

Examples

Signal and Continue

Monitors in Java

Intrinsic Locks

Visibility

Explicit Locks

Monitor that Defines a Semaphore

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

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         if (isFull())
8             notFull.wait();
9         data[begin] = o;
10        begin = next(begin);
11        notEmpty.signal()
12    }
13
14    public Object take() {
15        if (isEmpty())
16            notEmpty.wait();
17        Object result = data[end];
18        end = next(end);
19        notFull.signal();
20        return result;
21    }
22
23    private boolean isEmpty() { return begin == end; }
24    private boolean isFull()   { return next(begin) == end; }
25    private int next(int i)    { return (i+1) % N; }
26 }
```

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         if (writers != 0 or not OKtoWrite.empty()) {
8             OKtoRead.wait();
9         }
10        readers = readers + 1;
11        OKtoRead.signal();
12    }
13
14    public void EndRead {
15        readers = readers - 1;
16        if (readers==0) {
17            OKtoWrite.signal();
18        }
19    }
20    // continues
```

- ▶ first blocked writer given precedence over waiting readers (l-7)
- ▶ cascading unblocking (l-11)

Readers/Writers

```
1  public void StartWrite() {
2      if (writers != 0 or readers != 0) {
3          OKtoWrite.wait();
4      }
5      writers = writers + 1;
6  }
7
8  public void EndWrite() {
9      writers = writers - 1;
10     if (OKtoRead.empty()) {
11         OKtoWrite.signal();
12     } else {
13         OKtoRead.signal();
14     };
15 }
16 }
```

No starvation:

- ▶ If there are blocked writers, a new reader is required to wait until termination of (at least) the first write
- ▶ If there are blocked readers, they will (all) be unblocked before the next write

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         if (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

Examples

Signal and Continue

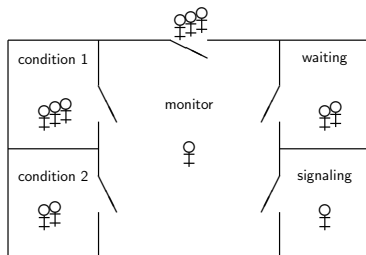
Monitors in Java

Intrinsic Locks

Visibility

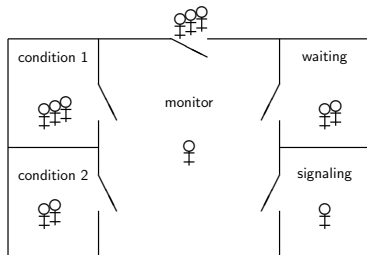
Explicit Locks

Strategies for Signal



- ▶ Two strategies for signal:
 - ▶ **Signal and Urgent Wait:** $E < S < W$ (classical monitors)
 - ▶ **Signal and Continue:** $E = W < S$ (Java)
- ▶ where
 - ▶ Precedence of signalling processes S
 - ▶ Precedence of waiting processes W
 - ▶ Precedence of processes blocked on entry E
- ▶ We already considered **Signal and Urgent Wait**
- ▶ We now consider **Signal and Continue**

Signal and Continue: $E = W < S$ (Java)



- ▶ Process from S which executes the signal continues execution
- ▶ Process from W which is unblocked joins competition for the lock
- ▶ Problem: signaling process can modify the condition after it executed the signal

Recall the Monitor that Defines a Semaphore

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

Incorrect under signal-and-continue

Signal and Continue

- ▶ Must re-check the condition

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

- ▶ Risk: introduce starvation.
- ▶ Seems less intuitive, but is preferred discipline today
 - ▶ Simpler formal semantics
 - ▶ Compatible with priority policies

Signal and Continue

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

Signal and Continue

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

- ▶ Is it fair?
- ▶ What happens with a process that is waiting to acquire the lock on the condition variable's queue?

Signal and Continue

- ▶ Not fair because a process outside the monitor could steal the permission
- ▶ Possible measure: pass the permission on to a blocked process on the condition variable
- ▶ This requires that the process that executes the signal detects whether there are blocked processes on the associated condition variable

Monitor That Defines a Fair Semaphore

```
1 public void acquire() {
2     if (permissions == 0) {
3         nonZero.wait();
4     } else {
5         permissions--;
6     }
7 }
8
9 public void release() {
10    if (nonZero.empty())
11        permissions++;
12    else // else case does not increment permissions
13        nonZero.signal();
14 }
```

Monitors

Examples

Signal and Continue

Monitors in Java

Intrinsic Locks

Visibility

Explicit Locks

Monitors in Java

- ▶ Every class has a lock and a unique **condition variable**
 - ▶ Pros: convenient encapsulation (lock and condition variable cannot be tampered with)
 - ▶ Cons: multiple condition variables benefit clarity and efficiency
- ▶ Methods `wait`, `notify` and `notifyAll` belong to the class `Object`
- ▶ Must use `synchronize` keyword in each method of the monitor
 - ▶ Guarantees mutual exclusion
 - ▶ Allows operations on condition variables to be invoked

Example: Fair Semaphore in Java

```
1 class Semaphore {
2     private int permissions;
3     private int waiting=0;
4
5     public Semaphore(int permissions) {
6         this.permissions = permissions;
7     }
8
9     public synchronized void acquire() throws InterruptedException {
10         if (permissions==0) {
11             waiting++;
12             wait();
13             waiting--;
14         } else {
15             permissions--;
16         }
17
18     public synchronized void release() {
19         if (waiting>0) {
20             notify();
21         } else {
22             permissions++;
23         }
24     }}
```

- ▶ Unexpected problem = 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 8 [1] (page 642)

Example: Fair Semaphore in Java

```
1 class Semaphore {
2     private int permissions;
3     private int waiting=0;
4     private int passedPermissions=0;
5
6     public Semaphore(int permissions) {
7         this.permissions = permissions;
8     }
9
10    public synchronized void acquire() throws InterruptedException {
11        while (permissions==0 && passedPermissions==0) {
12            waiting++;
13            wait();
14            waiting--;
15        }
16        if (passedPermissions>0) {
17            passedPermissions--;
18        } else {
19            permissions--;
20        }
21    }
22
23    public synchronized void release() {
24        if (waiting>0) {
25            notify();
26            passedPermissions++;
27        } else {
28            permissions++;
29        }
30    }
31 }
```

IllegalMonitorStateException

- ▶ Methods `wait`, `notify` and `notifyAll` can only be called from synchronized methods
- ▶ Otherwise an `IllegalMonitorStateException` is raised

```
1 public void m1() {  
2     this.wait();    //IllegalMonitorStateException  
3 }  
4  
5 public void m2() {  
6     this.notify();  //IllegalMonitorStateException  
7 }
```

Intrinsic Locks are Reentrant

- ▶ Reentrancy means locks are acquired on a per thread basis (not per invocation)
- ▶ Crucial for code like this not to deadlock

```
1 public class A {  
2     public synchronized void doSomething() {  
3         ...  
4     }  
5 }  
6  
7 public class B extends A {  
8     public synchronized void doSomething() {  
9         super.doSomething();  
10 }
```

Monitors

Examples

Signal and Continue

Monitors in Java

Intrinsic Locks

Visibility

Explicit Locks

Visibility

- ▶ Whether a thread can see the modifications of other threads
- ▶ `synchronization` also helps with visibility
 - ▶ It is not just for atomicity
- ▶ Visibility is subtle because the compiler may
 - ▶ Reorder operations
 - ▶ Cache values in registers

Example: sharing variables without synchronization

```
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: sharing variables without synchronization

```
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!

Volatile

- ▶ Operations on volatile variables are not reordered.
- ▶ Volatile variables are not cached: a read to a volatile variable always returns the most recent write by any thread

```
1 public class NoVisibility {  
2     volatile private static boolean ready;  
3     volatile private static int number;  
4     ...  
5 }
```

Note: can't use volatile to make assignment on shared variable atomic (e.g. `count++`)

Monitors

Examples

Signal and Continue

Monitors in Java

Intrinsic Locks

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 }
```