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

```
class Counter {
   private int c = 0;
   private Semaphore mutex = new Semaphore(1);

   public void inc() {
      mutex.acquire();
      c++;
      mutex.release()
   }
   public void dec() {
      mutex.acquire();
      c--;
      mutex.release();
   }
}
```

Counter using Monitors

```
monitor Counter {
  private int counter = 0;

  public void inc() {
    counter++;
  }

  public void dec() {
    counter--;
  }

}
```


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

Counter in Java

```
class Counter {

  private int counter = 0;

  public synchronized void inc() {
    counter++;
  }

  public synchronized void dec() {
    counter--;
  }

}
```


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

```
monitor Buffer {
  private Object buffer = null; // shared buffer
  private condition full;  // wait until space available
  private condition empty; // wait until buffer available

  public Object consume() {
    while (buffer == null)
      full.wait();
    Object aux = buffer;
    buffer = null;
    empty.signal();
    return aux;
  }

  public void produce(Object o) {
    while (buffer != null)
      empty.wait();
    buffer = o;
    full.signal();
  }

}
```

Example: Buffer of Size 1 in Groovy

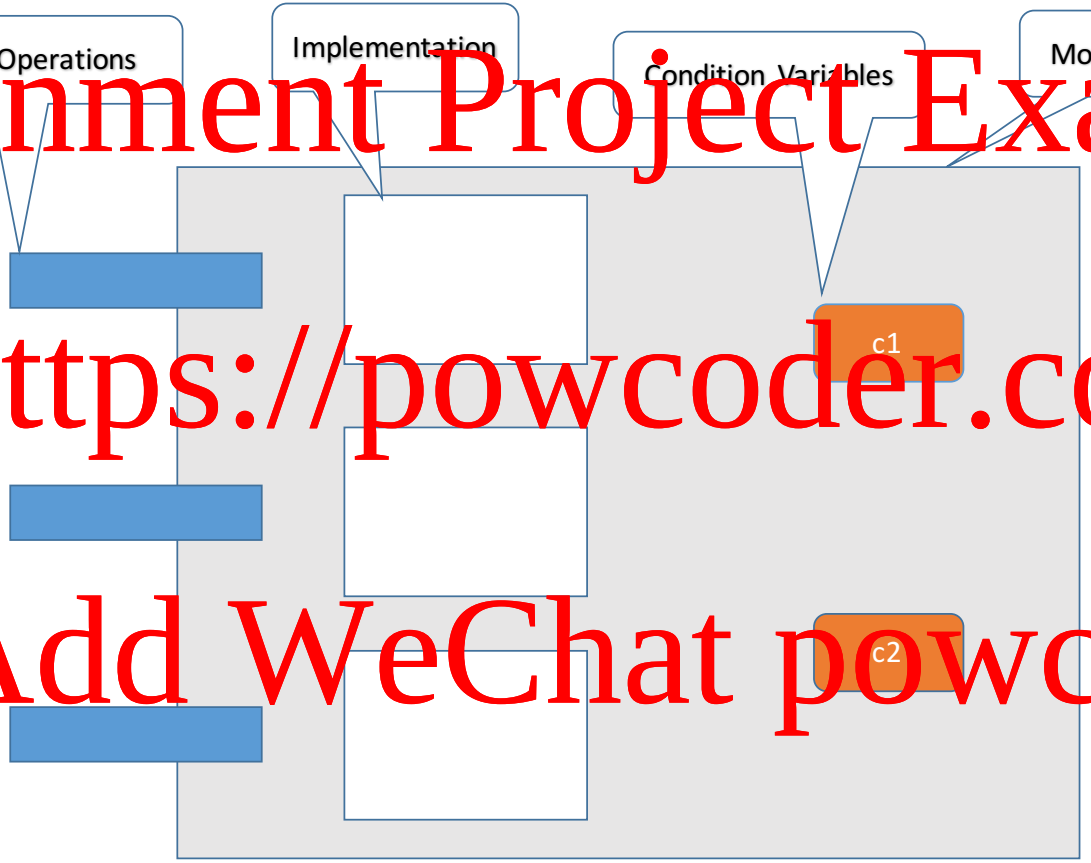
```
class Buffer {
  Object buffer = null; // shared buffer

  synchronized Object consume() {
    while (buffer == null)
      wait(); // wait on object's wait-set
    Object aux = buffer;
    buffer = null;
    signalAll(); // signal on object's wait-set
    return aux;
  }

  synchronized void produce(Object o) {
   while (buffer != null)
      wait();  // wait on object's wait-set
    buffer = o;
    signalAll(); // signal on object's wait-set
  }
}
```


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

Example: Buffer of Size 1 in Groovy (cont)

```
Buffer b =new Buffer();

20.times { // 20 producers
  int id=it
  Thread.start {
    b.produce(id);
  }
}

Thread.start { // Consumer 1
  println "consumer "+ b.consume();
}

Thread.start { // Consumer 2
  println "consumer "+ b.consume();
}

return ;
```

Explaining Monitors Graphically



Typical Behavior

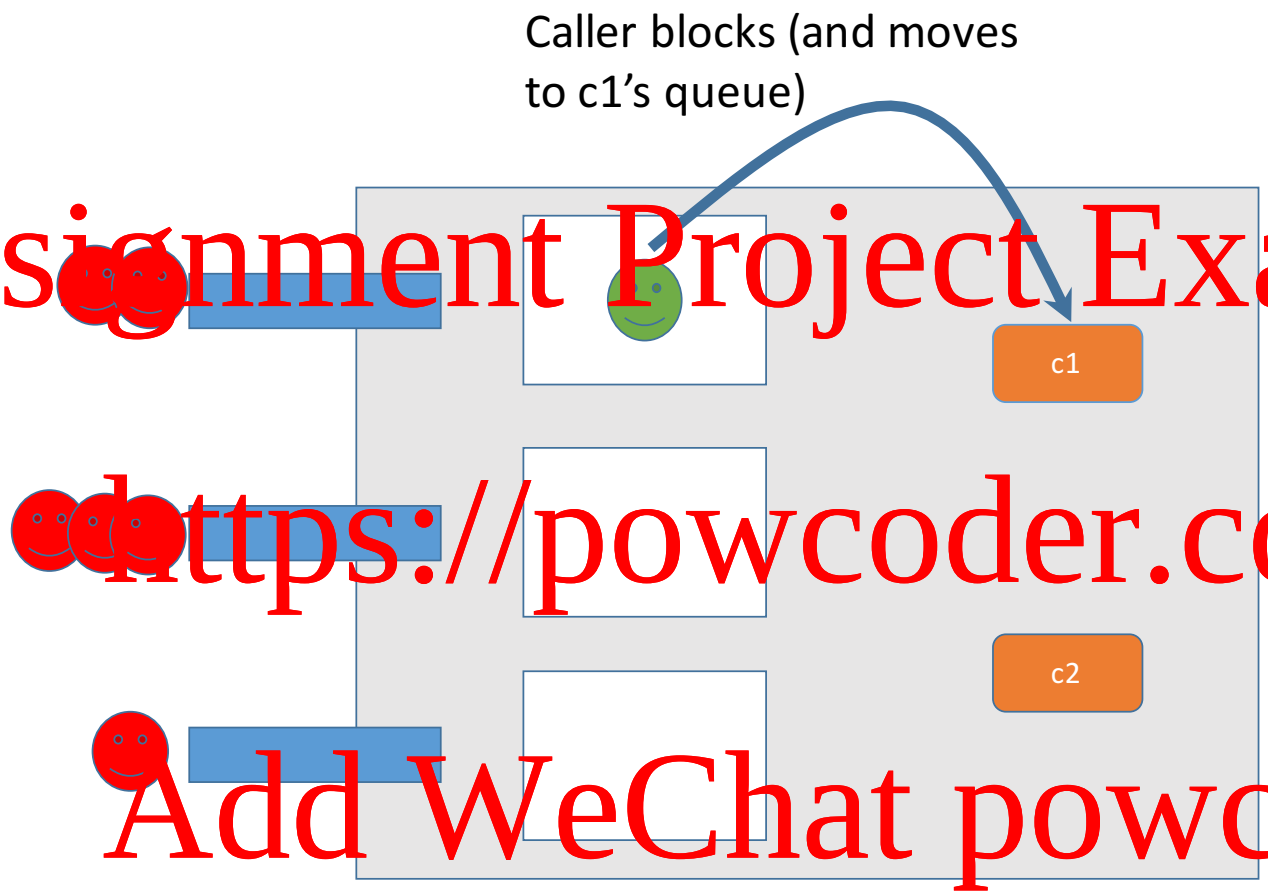
Typical Behavior

Typical Behavior

Typical Behavior

Typical Behavior

Wait



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

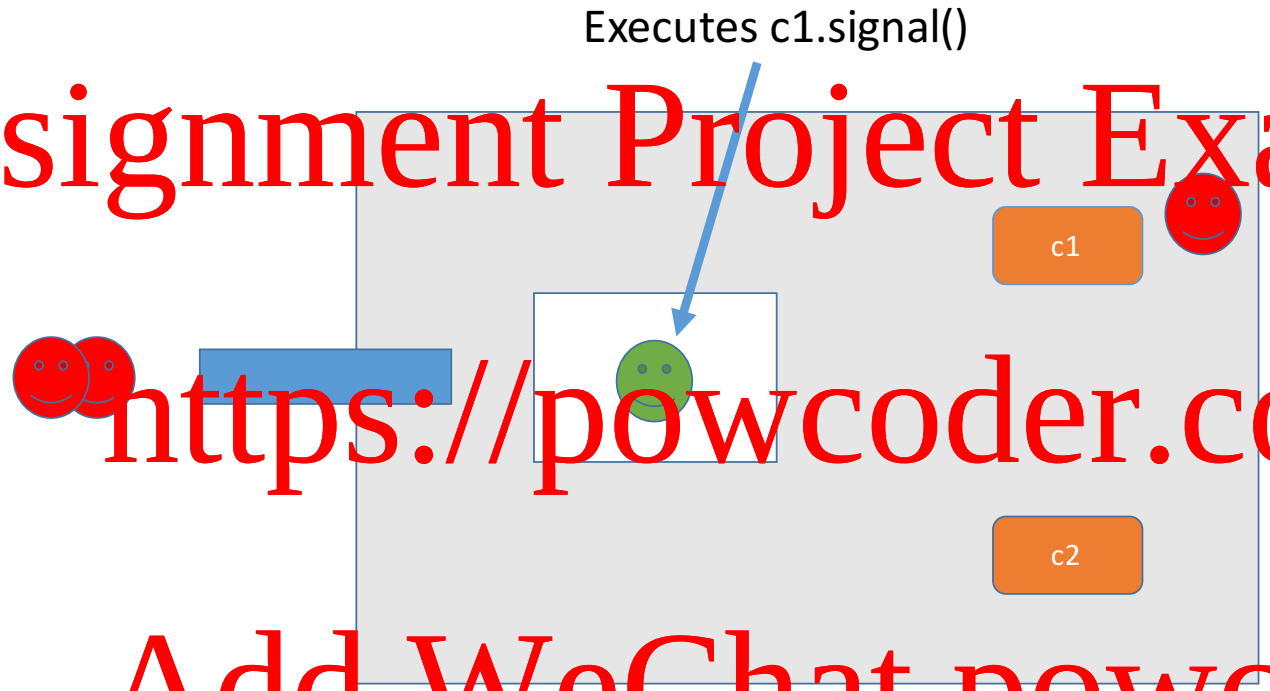
Wait

- 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

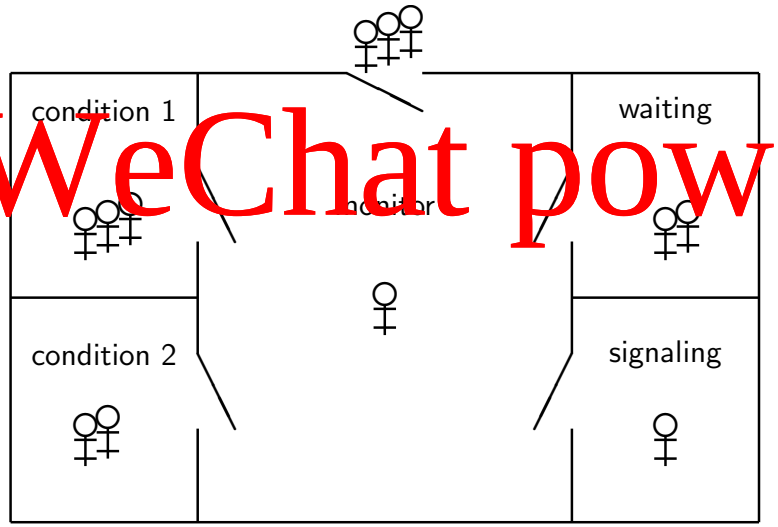
Signal



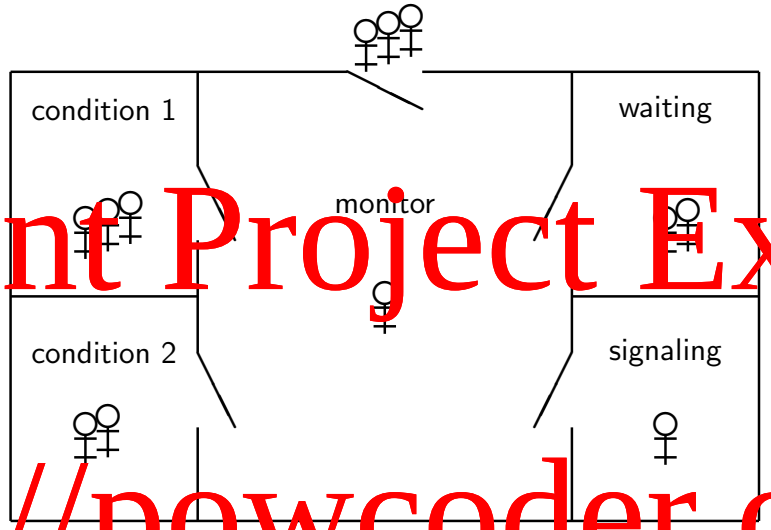
- Signalling process continues to execute after notifying on `c1`?
- Processes waiting in `c1`'s queue start immediately running inside the monitor?
- What about the processes blocked on entry to the monitor?

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



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

Monitor that Defines a Semaphore

```
monitor Semaphore {
  private condition nonZero;
  private int permissions;

  public Semaphore(int n) {
    this.permissions = n;
  }

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

  public void release() {
    permissions++;
    nonZero.notifyAll();
  }

}
```

Notify and Continue

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


```
public 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]

[1] JLS for Java 8 [1] (page 642)

Signal and Continue

```
monitor Semaphore {
  private condition nonZero;
  private int permissions;

  public Semaphore(int n) {
    this.permissions = n;
  }

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

  public void release() {
    permissions++;
    nonZero.signal();
  }
}
```


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

Buffer of Size *n*

```
monitor Buffer {
  private Object[] data = new Object[N];
  private int begin = 0, end = 0;
  private condition notFull, notEmpty;

  public void put(Object o) {
    while (isFull())
        notFull.wait();
    data[begin] = o;
    begin = (begin+1) % N;
    notEmpty.notify();
  }

  public Object take() {
    while (isEmpty())
        notEmpty.wait();
    Object result = data[end];
    end = (end+1) % N
    notFull.notify();
    return result;
  }

  private boolean isEmpty() { return begin == end; }
  private boolean isFull()  { return next(begin) == end; }
}
```

Readers/Writers


```
monitor RW {
  ...

  public void read() {
    ...
  }

  public void write() {
    ...
  }

}
```


What is the problem with this setting?

Readers/Writers

```
monitor RW {
  int readers = 0;
  int writers = 0;
  condition OKtoRead, OKtoWrite;

  public void StartRead() {
    while (writers != 0 or not OKtoWrite.empty()) {
      OKtoRead.wait();
    }
    readers = readers + 1;
  }

  public void EndRead {
    readers = readers - 1;
    if (readers==0) {
      OKtoWrite.notify();
    }
  }
  // continues
```

Readers/Writers

```
  public void StartWrite() {
    while (writers != 0 or readers != 0) {
      OKtoWrite.wait();
    }
    writers = writers + 1;
  }

  public void EndWrite() {
    writers = writers - 1;
    OKtoWrite.signal();
    OKtoRead.signalAll();
  }
}
```

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 queuing for entering the monitor, the waiting writers will never execute

Dining Philosophers

```
monitor ForkMonitor {
  int[] fork = {2,2,2,2,2};
  condition[] OKtoEat; // 0-4

  public void takeForks(integer i) {
    while (fork[i] != 2) {
      OKtoEat[i].wait();
    }
    fork[i+1] = fork[i+1] - 1;
    fork[i-1] = fork[i-1] - 1;
  }

  public void releaseForks(integer i) {
    fork[i+1] = fork[i+1] + 1;
    fork[i-1] = fork[i-1] + 1;
    if (fork[i+1] == 2) {
      OKtoEat[i+1].signal();
    }
    if (fork[i-1] == 2) {
      OKtoEat[i-1].signal();
    }
  }
}
```

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

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


```
int n=0;
Thread.start { // P
  int local1, local2;
  n = some expression;
  computation not using n;
  local1 = (n+5)*7;
  local2 = n+5;
  n = local1 *  local2;
}
```

```
Thread.start { // Q
  int local;
  local = n+6;


}
```


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
2 computation not using n
3 tempReg2 = tempReg1 + 5
4 local2 = tempReg2
5 local1 = tempReg2 * 7
6 n = local1 * local2
```

```
1 n = some expression;
2 computation not using n;
3 local1 = (n+5)*7
4 local2 = n+5;
5 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 1[2]

```
1  public class SharedVariable {
2     private static int  sharedVariable = 0;
3     public static void main(String[] args) throws InterruptedExcept
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

```java
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```


- `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!

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

```
public abstract class BaseBoundedBuffer <V> {
    private final V[] buf;
    private int tail;
    private int head;
    private int count;

    protected BaseBoundedBuffer(int capacity) {
        this.buf = (V[]) new Object[capacity];
    }

    protected synchronized final void doPut(V v) {
        buf[tail] = v;
        if (++tail == buf.length)
            tail = 0;
        ++count;
    }

   // continued
```

Bounded Buffers Revisited

```
    protected synchronized final V doTake() {
        V v = buf[head];
        buf[head] = null;
        if (++head == buf.length)
            head = 0;
        --count;
        return v;
    }

    public synchronized final boolean isFull() {
        return count == buf.length;
    }

    public synchronized final boolean isEmpty() {
        return count == 0;
    }
}
```

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

```
    // BLOCKS-UNTIL: not-empty
    public synchronized V take() throws InterruptedException {
        while (isEmpty())
            wait();
        V v = doTake();
        notifyAll();
        return v;
    }

    // BLOCKS-UNTIL: not-full
    // Alternate form of put() using conditional notification
    public synchronized void alternatePut(V v) throws InterruptedE
        while (isFull())
            wait();
        boolean wasEmpty = isEmpty();
        doPut(v);
        if (wasEmpty)
            notifyAll();
    }
}
```

Lock Interface

```
public interface Lock {
  void lock();                          // Acquires the lock.
  void lockInterruptibly() throws InterruptedException;
                                        // Acquires the lock unless
                                        // the current thread is interrupted.
  boolean tryLock();                    // Acquires the lock only if it is
                                        // free at the time of invocation.
  boolean tryLock(long time, TimeUnit unit)
    throws InterruptedException; // Acquires the lock if it
          // is free within the given waiting time
          // and the current thread has not been interrupted.
  void unlock();                        // Releases the lock.
  Condition newCondition();             // Returns a new Condition instance
                                        // that is bound to this Lock instance.
}
```

Using Locks


```
Lock l = new ReentrantLock();
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

Using Condition Interface

```
public interface Condition {
  void await(); // Causes the current thread to wait until
                // it is signalled or interrupted.
  boolean await(long time, TimeUnit unit);
  long awaitNanos(long nanosTimeout);
  void awaitUninterruptibly();
  boolean awaitUntil(Date deadline);

  void signal();     // Wakes up one waiting thread.
  void signalAll(); // Wakes up all waiting threads.
}
```

- 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

```
    // BLOCKS-UNTIL: notEmpty
    public T take() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            T x = items[head];
            items[head] = null;
            if (++head == items.length)
                head = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```