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

```
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 Hydra nor Java)

Counter in Java

```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
- 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.

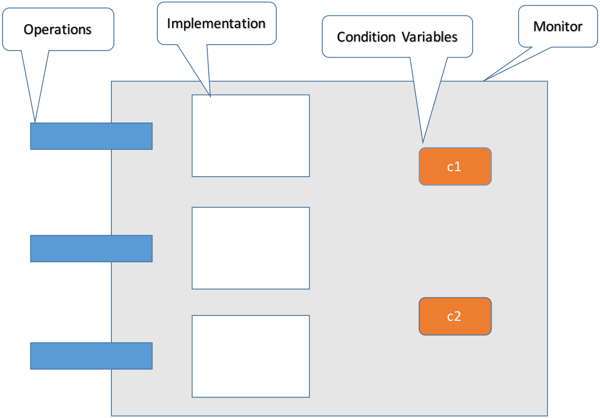
`Cond.empty()`

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

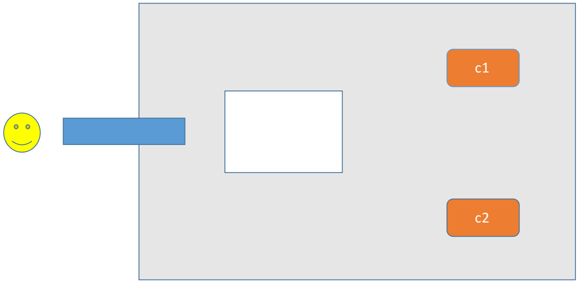
Example: Buffer of Size 1

```
monitor Buffer {
  private condition notFull;  // wait until space available
  private condition notEmpty; // wait until data available

  private Object data = null; // shared data

  public Object read() {
    if (data == null)
      notEmpty.wait();
    aux = data;
    data = null;
    notFull.signal();
    return aux;
  }

  public void write(Object o) {
    if (data != null)
      notFull.wait();
    data = o;
    notEmpty.signal();
  }

}
```

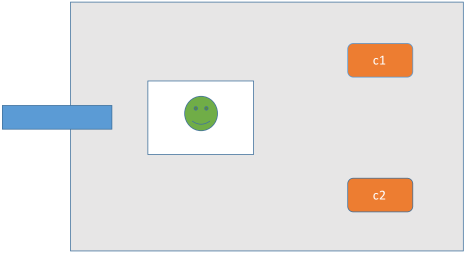
Explaining Monitors Graphically



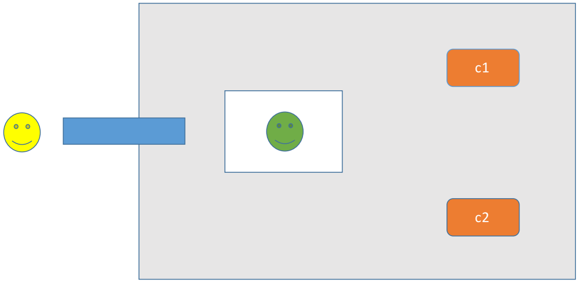
Typical Behavior



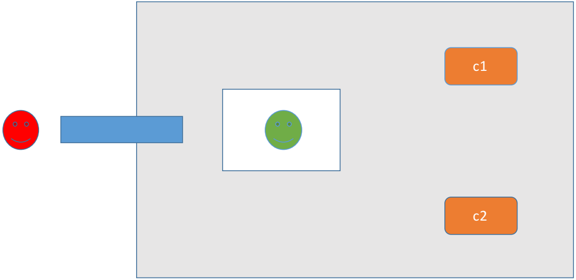
Typical Behavior



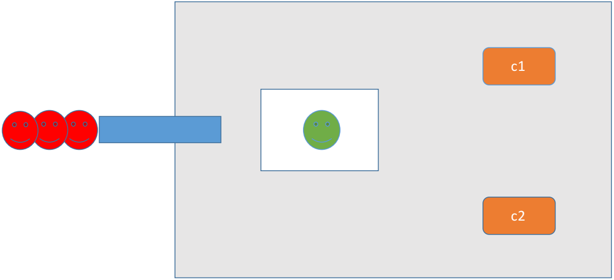
Typical Behavior



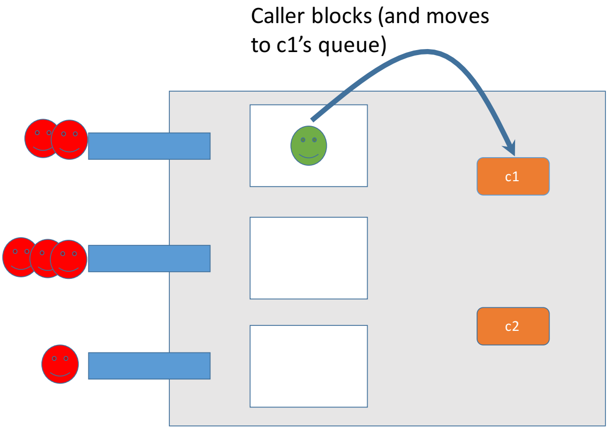
Typical Behavior



Typical Behavior

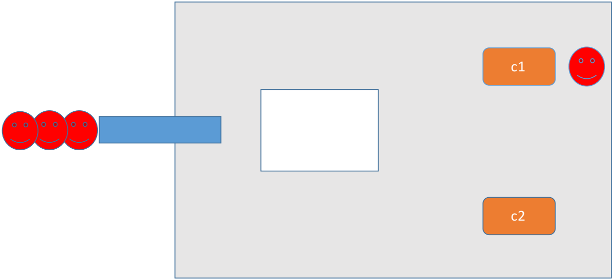


Wait



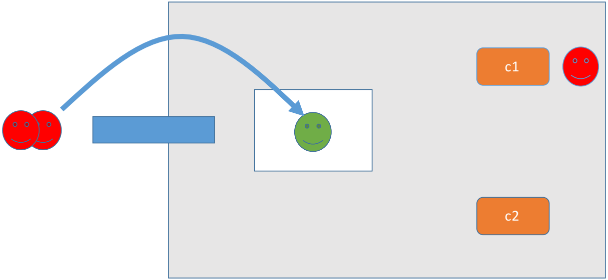
- Blocks process currently executing and associates it to variable's queue
- Upon blocking frees the lock allowing the entry of other processes

Wait



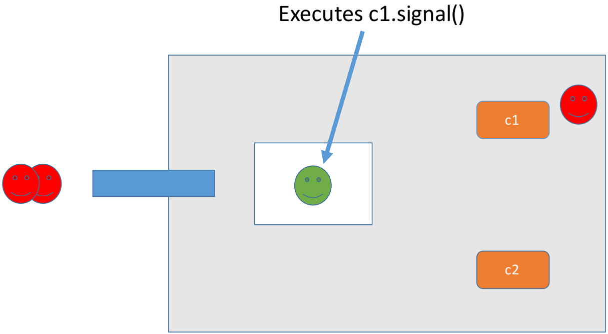
- Blocks process currently executing and associates it to variable's queue
- Upon blocking frees the lock allowing the entry of other processes

Wait



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

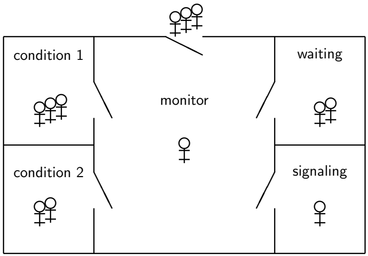
Signal



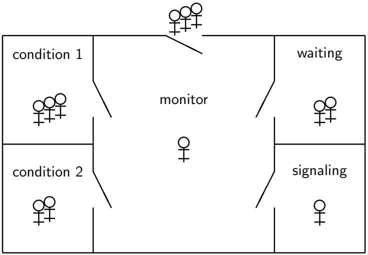
- Signalling process continues to execute after signalling on `c1`?
- Processes waiting in `c1`'s queue start immediately running inside the monitor?
- What about the processes blocked on entry to the monitor?

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

```
monitor Buffer {
  private condition notFull;  // wait until space available
  private condition notEmpty; // wait until data available

  private Object data = null; // shared data

  public Object read() {
    if (data == null)
      notEmpty.wait();
    aux = data;
    data = null;
    notFull.signal();
    return aux;
  }

  public void write(Object o) {
    if (data != null)
      notFull.wait();
    data = o;
    notEmpty.signal();
  }

}
```

Monitor that Defines a Semaphore

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

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

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

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

}
```

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) {
    if (isFull())
         notFull.wait();
    data[begin] = o;
    begin = next(begin);
    notEmpty.signal()
  }

  public Object take() {
    if (isEmpty())
       notEmpty.wait();
    Object result = data[end];
    end = next(end);
    notFull.signal();
    return result;
  }

  private boolean isEmpty() { return begin == end; }
  private boolean isFull()  { return next(begin) == end; }
  private int next(int i)   { return (i+1) % N; }
}
```

Readers/Writers

```
monitor RW {
  ...

  public void read() {
    ...
  }

  public void write() {
    ...
  }

}
```

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

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

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

No starvation:

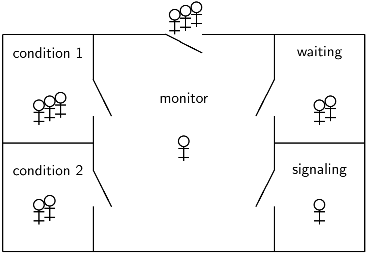
- If there are blocked writers, a new reader is required to wait until termination of (at least) the first write
- If there are blocked readers, they will (all) be unblocked before the next write

Dining Philosophers

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

  public void takeForks(integer i) {
    if (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();
    }
  }
}
```

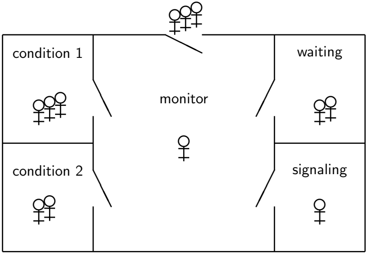
`forks[i]` is number of forks available to philosopher *i*

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

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

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

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

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

}
```

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

```
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();
  }
}
```

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?

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

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

public void release() {
  if (nonZero.empty())
    permissions++;
  else // else case does not increment permissions
    nonZero.signal();
}
```

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

```
class Semaphore {
  private int permissions;
  private int waiting=0;

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

  public synchronized void acquire() throws InterruptedException {
      if (permissions==0) {
          waiting++;
          wait();
          waiting--;
      } else {
          permissions--;}
  }

  public synchronized void release() {
      if (waiting>0) {
          notify();
      } else {
          permissions++;
      }
  }}
```

- 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]

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

Example: Fair Semaphore in Java

```java
class Semaphore {
  private int permissions;
  private int waiting=0;
  private int passedPermissions=0;

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

  public synchronized void acquire() throws InterruptedException {
      while (permissions==0 && passedPermissions==0) {
          waiting++;
          wait();
          waiting--;
      }
      if (passedPermissions>0) {
          passedPermissions--;
      } else {
          permissions--;
      }
  }

  public synchronized void release() {
      if (waiting>0) {
          notify();
          passedPermissions++;
      } else {
          permissions++;
      }
  }
}
```

IllegalMonitorStateException

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

```java
public void m1() {
  this.wait();     //IllegalMonitorStateException
}

public void m2() {
  this.notify();   //IllegalMonitorStateException
}
```

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

```java
public class A {
  public synchronized void doSomething() {
    ...
  }
}

public class B extends A {
  public synchronized void doSomething() {
    super.doSomething();
}
```

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

```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?

Example: sharing variables without synchronization

```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!

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

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

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

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

```java
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

```java
public class BoundedBuffer <V> extends BaseBoundedBuffer<V> {
    // CONDITION PREDICATE: not-full (!isFull())
    // CONDITION PREDICATE: not-empty (!isEmpty())
    public BoundedBuffer() {
        this(100);
    }

    public BoundedBuffer(int size) {
        super(size);
    }

    // BLOCKS-UNTIL: not-full
    public synchronized void put(V v) throws InterruptedException
        while (isFull())
            wait();
        doPut(v);
        notifyAll();
    }
    // 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

```java
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

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

Using Condition Interface

```java
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

```java
public class ConditionBoundedBuffer <T> {
    protected final Lock lock = new ReentrantLock();
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();
    private static final int BUFFER_SIZE = 100;
    private final T[] items = (T[]) new Object[BUFFER_SIZE];
    private int tail, head, count;

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