Busy-wait solution to the bounded buffer problem:

```
index: 0..SIZE-1
buf: array [index] of data
nextempty, nextfull, fullslots : index := 0, 0, 0
mutex : spinlock

procedure insert(d : data)
   loop
      acquire(mutex)
      if fullslots < SIZE
         buf[nextempty] := d
         nextempty++; nextempty %= SIZE
         fullslots++
         release(mutex)
         return
      else
         release(mutex)

procedure remove : data
   loop
      acquire(mutex)
      if fullslots > 0
         data d := buf[nextfull];
         nextfull++; nextfull %= SIZE;
         fullslots--;
         release(mutex)
         return d
      else
         release(mutex)
```

Semaphore-based solution:

```
shared buf : array [1..SIZE] of data
shared next_full, next_empty : integer := 1
shared mutex : semaphore := 1
shared empty_slots, full_slots : semaphore := SIZE, 0

procedure insert(d : data) :
   P(empty_slots)
   P(mutex)
   buf[next_empty] := d
   next_empty := next_empty mod SIZE + 1
   V(mutex)
   V(full_slots)

function remove returns data :
   P(full_slots)
   P(mutex)
   d : data := buf[next_full]
   next_full := next_full mod SIZE + 1
   V(mutex)
   V(empty_slots)
   return d
```

Java 2 monitor solution with a single condition queue:

```
class BB {
   final private int SIZE = 10;
   private Object[] buf = new Object[SIZE];

   private int nextEmpty = 0;
   private int nextFull = 0;
   private int fullSlots = 0;

   synchronized public void insert(Object d)
         throws InterruptedException {
      while (fullSlots == SIZE) {
         wait();
      }
      buf[nextEmpty] = d;
      nextEmpty = (nextEmpty + 1) % SIZE;
      ++fullSlots;
      notifyAll();       // explain why!
   }

   synchronized public Object remove()
         throws InterruptedException {
      while (fullSlots == 0) {
         wait();
      }
      Object d = buf[nextFull];
      nextFull = (nextFull + 1) % SIZE;
      --fullSlots;
      notifyAll();       // explain why!
      return d;
   }
}
```

Java 2 solution without using `notifyAll`

```
class BB {
   final private int SIZE = 10;
   private Object[] buf = new Object[SIZE];

   private Object producerMutex = new Object();
   // waited upon only by producers; protects the following:
      private int nextEmpty = 0;
      private int emptySlots = SIZE;

   private Object consumerMutex = new Object();
   // waited upon only by consumers; protects the following:
      private int nextFull = 0;
      private int fullSlots = 0;

   public void insert(Object d) throws InterruptedException {
      synchronized (producerMutex) {
         while (emptySlots == 0) {
            producerMutex.wait();
         }
         --emptySlots;
         buf[nextEmpty] = d;
         nextEmpty = (nextEmpty + 1) % SIZE;
      }
      synchronized (consumerMutex) {
         ++fullSlots;
         consumerMutex.notify();
      }
   }

   public Object remove() throws InterruptedException {
      Object d;
      synchronized (consumerMutex) {
         while (fullSlots < 0) {
            consumerMutex.wait();
         }
         --fullSlots;
         d = buf[nextFull];
         nextFull = (nextFull + 1) % SIZE;
      }
      synchronized (producerMutex) {
         ++emptySlots;
         producerMutex.notify();
      }
      return d;
   }
}
```

Java 5 solution, with multiple condition queues

```
class BB {
   final private int SIZE = 10;
   private Object[] buf = new Object[SIZE];

   private int nextEmpty = 0;
   private int nextFull = 0;
   private int fullSlots = 0;

   Lock l = new ReentrantLock();
   final Condition emptySlot  = l.newCondition();
   final Condition fullSlot = l.newCondition();

   public void insert(Object d) throws InterruptedException {
      l.lock();
      try {
         while (fullSlots == SIZE) {
            emptySlot.await();
         }
         buf[nextEmpty] = d;
         nextEmpty = (nextEmpty + 1) % SIZE;
         ++fullSlots;
         fullSlot.signal();
      } finally {
         l.unlock();
      }
   }

   public Object remove() throws InterruptedException {
      l.lock();
      try {
         while (fullSlots == 0) {
            fullSlot.await();
         }
         Object d = buf[nextFull];
         nextFull = (nextFull + 1) % SIZE;
         --fullSlots;
         emptySlot.signal();
         return d;
      } finally {
         l.unlock();
      }
   }
}
```