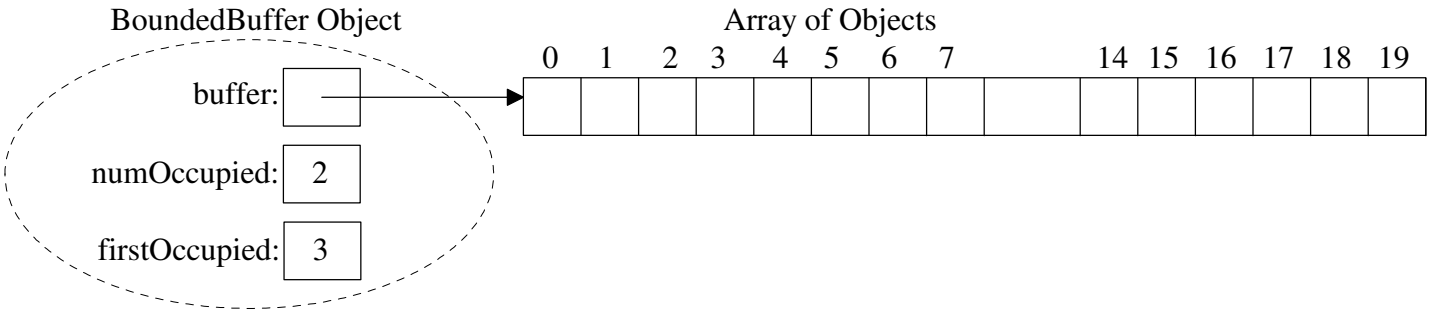
1. Consider the Java `BoundedBuffer` code.

a) Look at the invariant comment to decide which buffer slots would be full if:



b) Why are the `while` loops around the `wait()` statements necessary for correct functioning of the code?

2. We could try to reduce the overhead of notifications by using `notify` instead of `notifyAll`. Before doing this, our textbook author suggests we check:

- There is no danger of waking too few threads. Either you have some way to know that only one is waiting, or you know that only one would be able to proceed, with the others looping back to waiting.
- There is no danger of waking the wrong thread. Either you have some way to know that only one is waiting, or you know that all are equally able to proceed.

a) Why is changing all the `notifyAll` to `notify` unsafe for this `BoundedBuffer` code?

b) One limitation of Java is that each object has only a single condition variable. In the BoundedBuffer example, all threads wait on the single condition variable. If we could have multiple condition variables with a monitor, threads waiting to `insert` into a full buffer could wait on one condition variable, while threads waiting to `retrieve` from an empty buffer could wait on another condition variable. This way the `insert` code could notify a thread waiting to `retrieve` from an empty buffer and vice versa. Would such a change (say using POSIX which can model this) allow us to use `notify` instead of `notifyAll` in the `BoundedBuffer` code?

3. Recall the *semaphores* mechanism, which is a more specialized (and older) tool which is basically an unsigned integer with operations restricted to:
   - initialized when it is created to programmer's specified nonnegative integer value
   - `release` (or V) operation - increment the semaphore by 1
   - `acquire` (or P) operation - try to decrement the semaphore by 1, but waits if the semaphore value is 0 since it cannot go negative. Once another thread has performed a `release` operation to increment the semaphore the a waiting thread can continue its `release` operation

a) If we wanted to use a semaphore as a mutex, then

- what should the semaphore's initial value be?
- what operation would we call to lock the "mutex"?
- what operation would we call to unlock the "mutex"?

4. Consider the Java implementation of a `BoundedBuffer` below.

```
import java.util.concurrent.Semaphore;

public class BoundedBuffer {
  private java.util.List<Object> buffer =
    java.util.Collections.synchronizedList
    (new java.util.LinkedList<Object>());

  private static final int SIZE = 20; // arbitrary

  private Semaphore occupiedSem = new Semaphore(0);
  private Semaphore freeSem = new Semaphore(SIZE);

  /* invariant: occupiedSem + freeSem = SIZE
     buffer.size() = occupiedSem
     buffer contains entries from oldest to youngest */

  public void insert(Object o) throws InterruptedException{
    freeSem.acquire();
    buffer.add(o);
    occupiedSem.release();
  }

  public Object retrieve() throws InterruptedException{
    occupiedSem.acquire();
    Object retrieved = buffer.remove(0);
    freeSem.release();
    return retrieved;
  }
}
```

Figure 4.18: Alternative `BoundedBuffer` class, using semaphores

a) When the buffer is full how is a thread attempting an `insert` stalled?

b) How is a thread waiting to perform an `insert` resumed?

c) When the buffer is full how is a thread attempting a `retrieve` stalled?

d) How is a thread waiting to perform a `retrieve` resumed?

e) Do we want `synchronized` on the `insert` and `retrieve` methods?

5. Generally, when are semaphores a good choice for synchronization mechanisms?

6. Consider pseudocode for transferring some `amount` of money from `sourceAccount` to `destinationAccount`:

```
lock sourceAccount.mutex
lock destinationAccount.mutex
sourceAccount.balance = sourceAccount - amount
destinationAccount = destinationAccount + amount
unlock sourceAccount.mutex
unlock destinationAccount.mutex
```

a) Describe how deadlock can occur if I try to transfer $100 to your account at the same time you try to transfer $1 to my account.

b) Consider the Dining Philosophers problem below

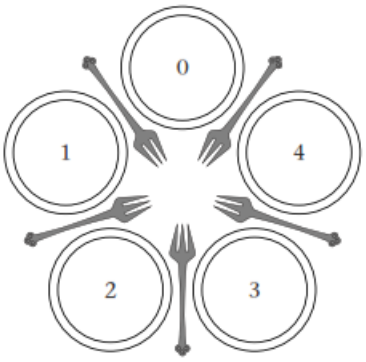


Figure 4.20: Five philosophers, numbered 0 through 4, have places around a circular dining table. There is a fork between each pair of adjacent places.

When can deadlock occur?

c) How might we prevent deadlock from occuring?

d) How might we detect deadlock has occurred?

e) How could we break a deadlock?