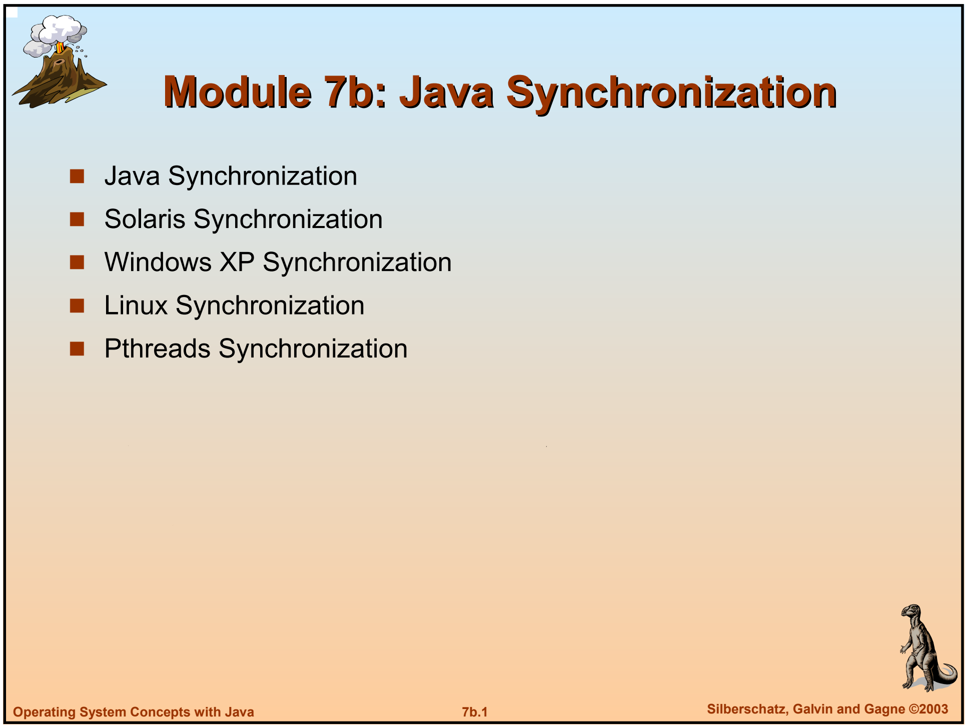
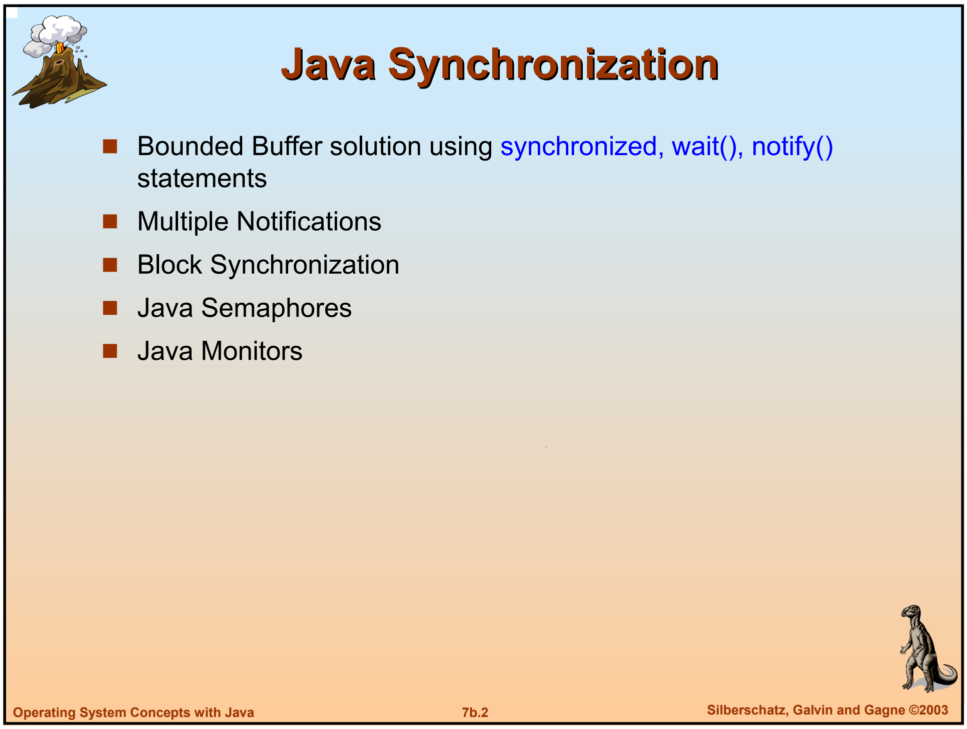


Module 7b: Java Synchronization

- Java Synchronization
- Solaris Synchronization
- Windows XP Synchronization
- Linux Synchronization
- Pthreads Synchronization



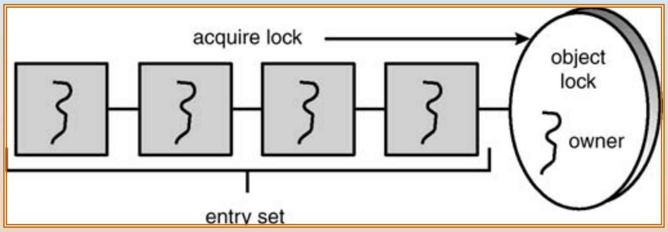
Java Synchronization

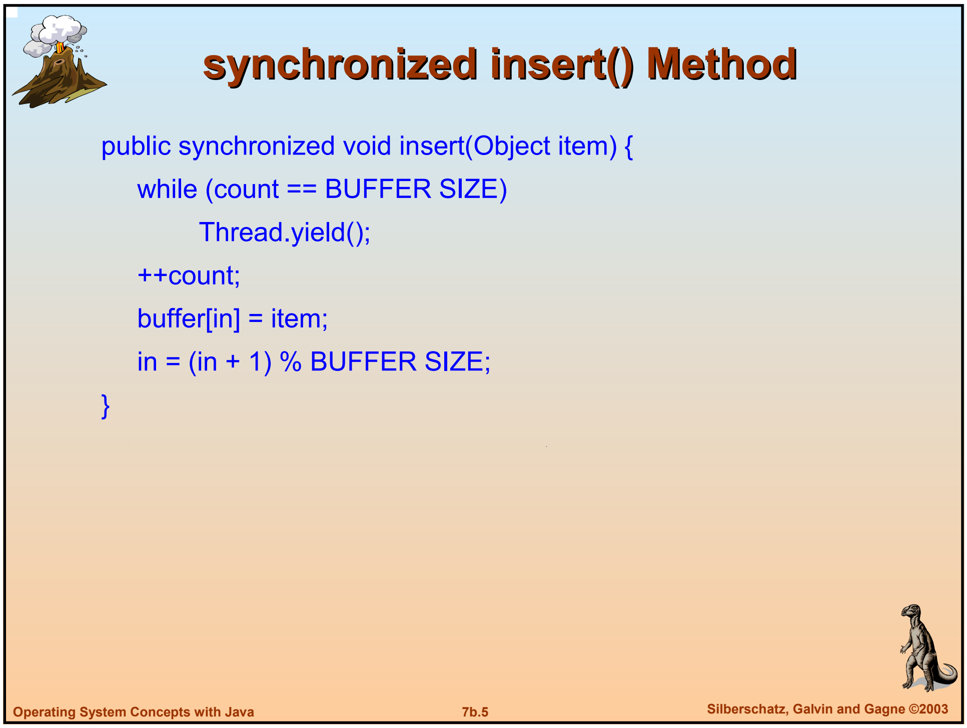
- Bounded Buffer solution using synchronized, wait(), notify() statements
- Multiple Notifications
- Block Synchronization
- Java Semaphores
- Java Monitors

synchronized Statement

- Every object has a lock associated with it
- Calling a synchronized method requires “owning” the lock
- If a calling thread does not own the lock (another thread already owns it), the calling thread is placed in the wait set for the object’s lock
- The lock is released when a thread exits the synchronized method

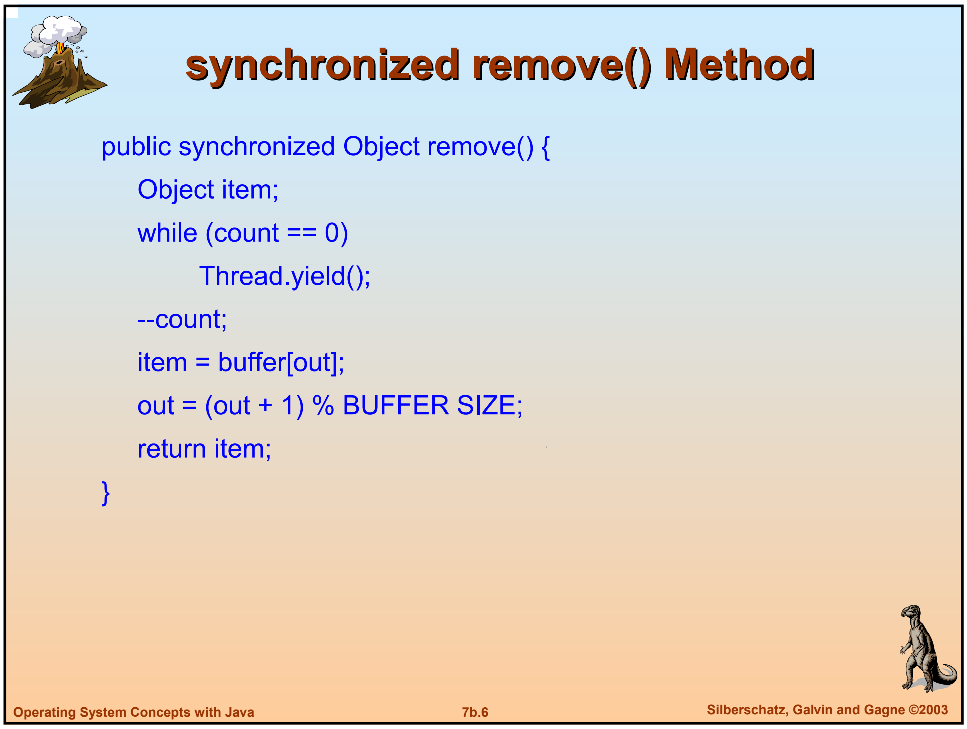
Entry Set





synchronized insert() Method

```
public synchronized void insert(Object item) {
    while (count == BUFFER SIZE)
          Thread.yield();
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```



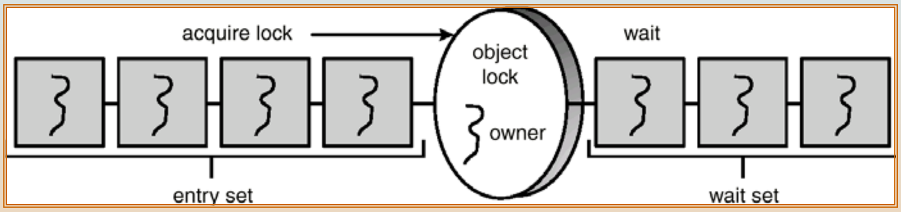
synchronized remove() Method

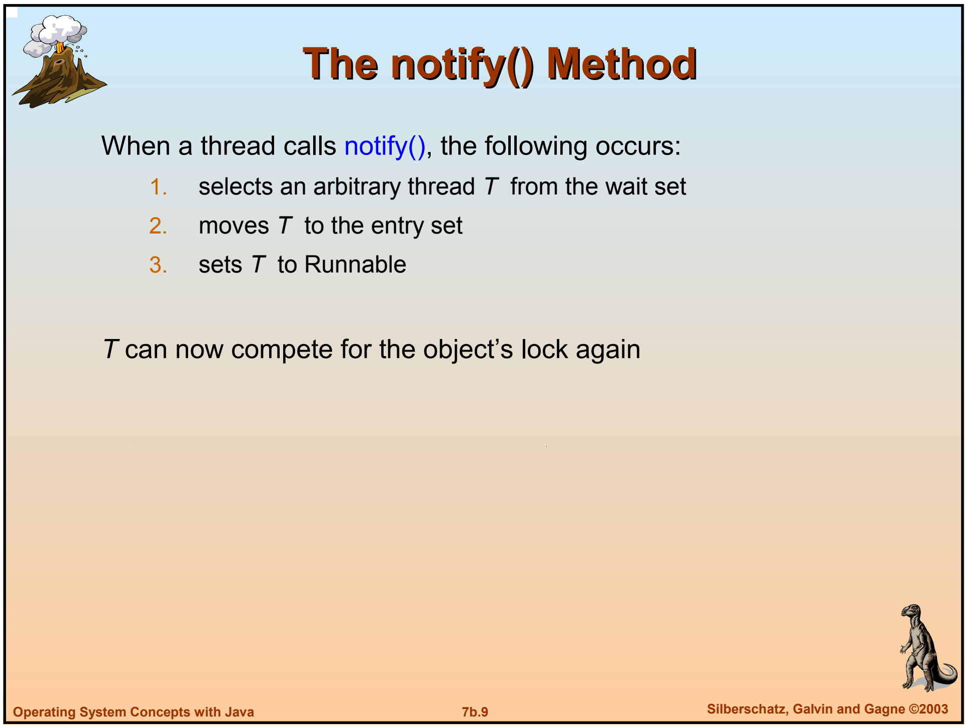
```
public synchronized Object remove() {
    Object item;
    while (count == 0)
          Thread.yield();
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    return item;
}
```

The wait() Method

- When a thread calls wait(), the following occurs:
  1. the thread releases the object lock
  2. thread state is set to blocked
  3. thread is placed in the wait set

Entry and Wait Sets



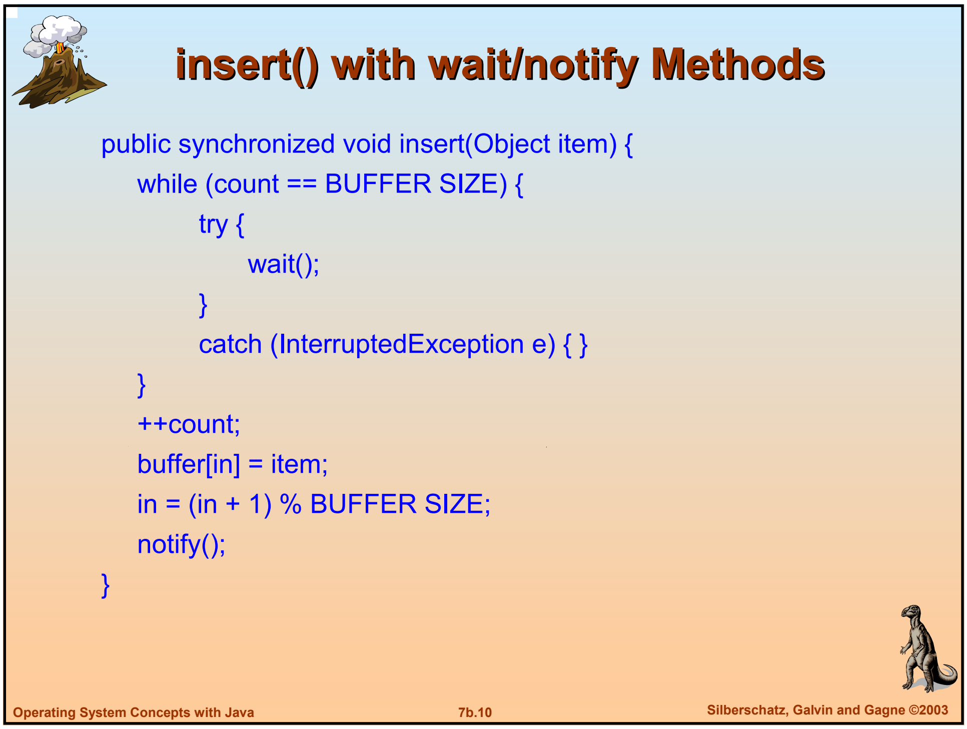


The notify() Method

When a thread calls notify(), the following occurs:

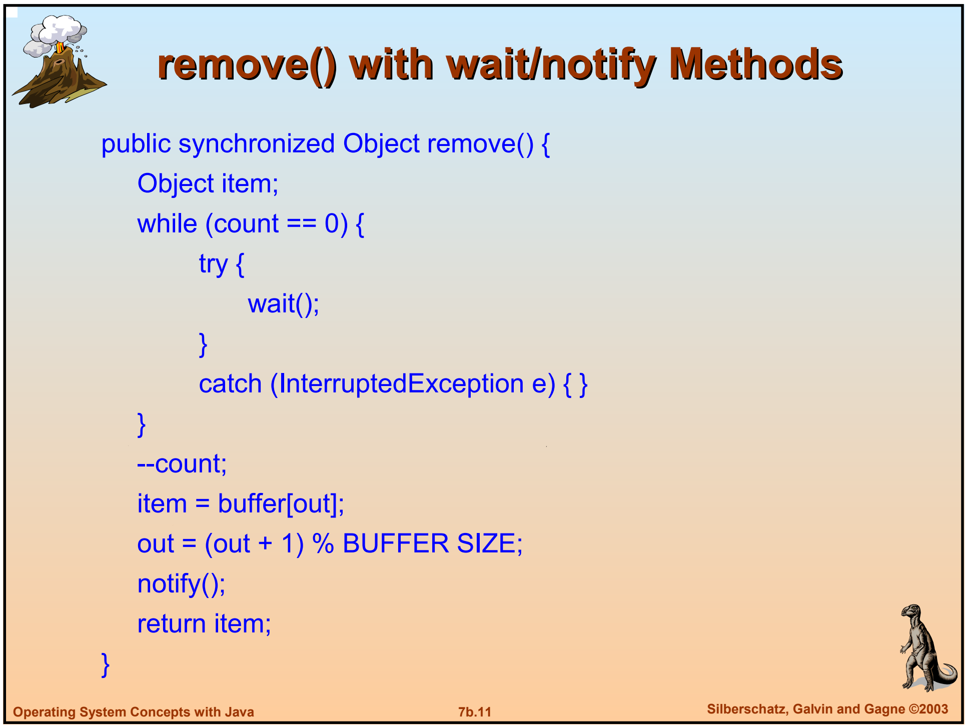
1. selects an arbitrary thread *T* from the wait set
2. moves *T* to the entry set
3. sets *T* to Runnable

T can now compete for the object’s lock again



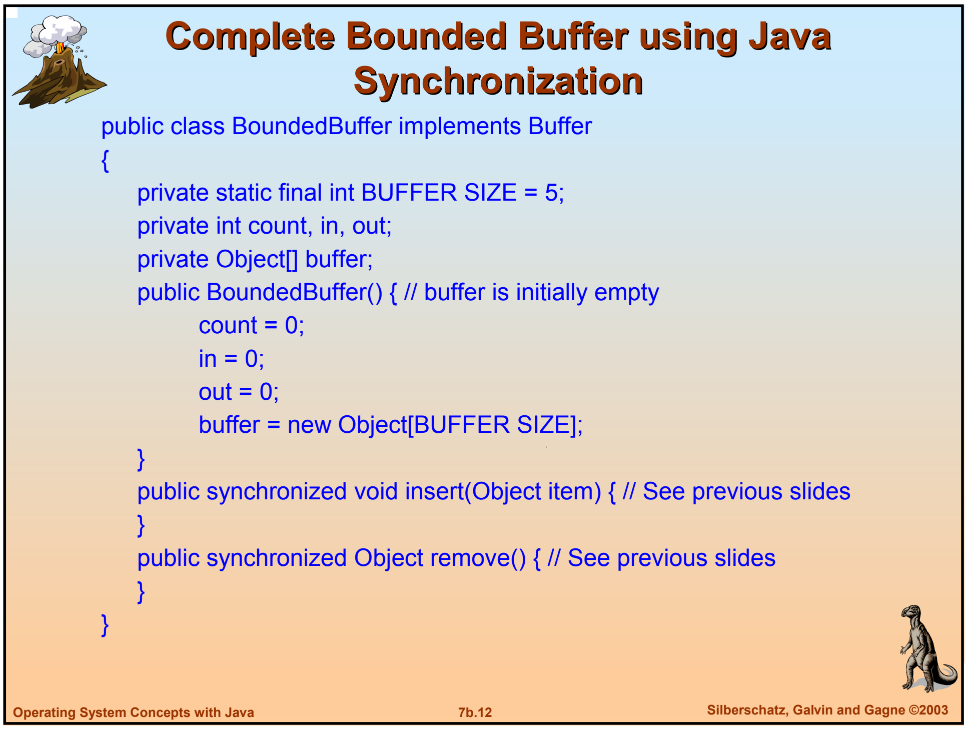
insert() with wait/notify Methods

```
public synchronized void insert(Object item) {
   while (count == BUFFER SIZE) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
   }
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
   notify();
}
```



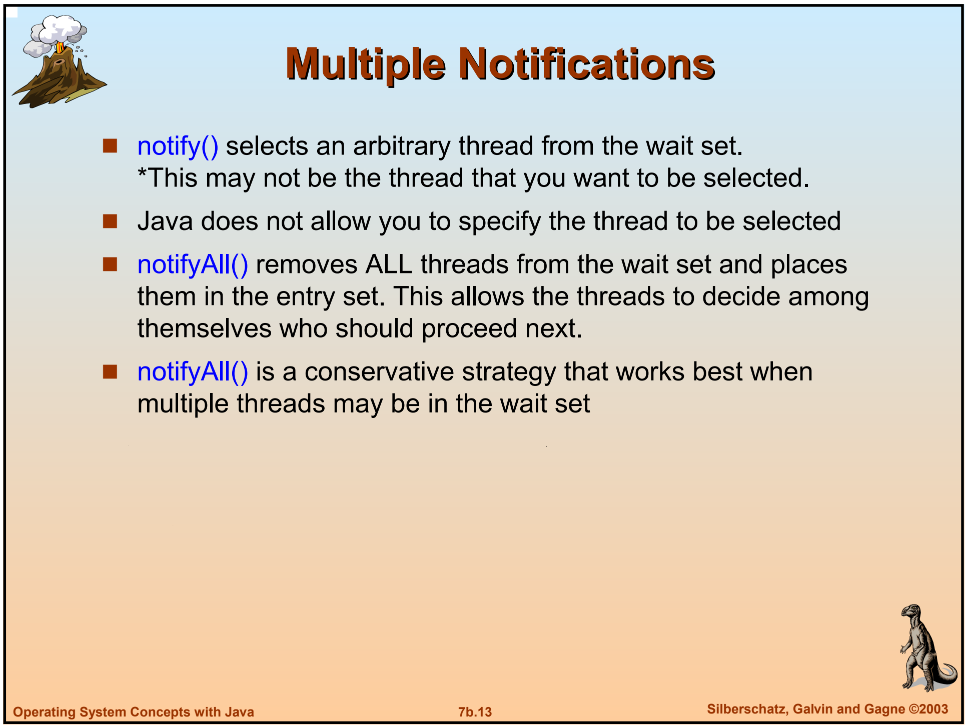
remove() with wait/notify Methods

```
public synchronized Object remove() {
    Object item;
    while (count == 0) {
          try {
               wait();
          }
          catch (InterruptedException e) { }
    }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    notify();
    return item;
}
```



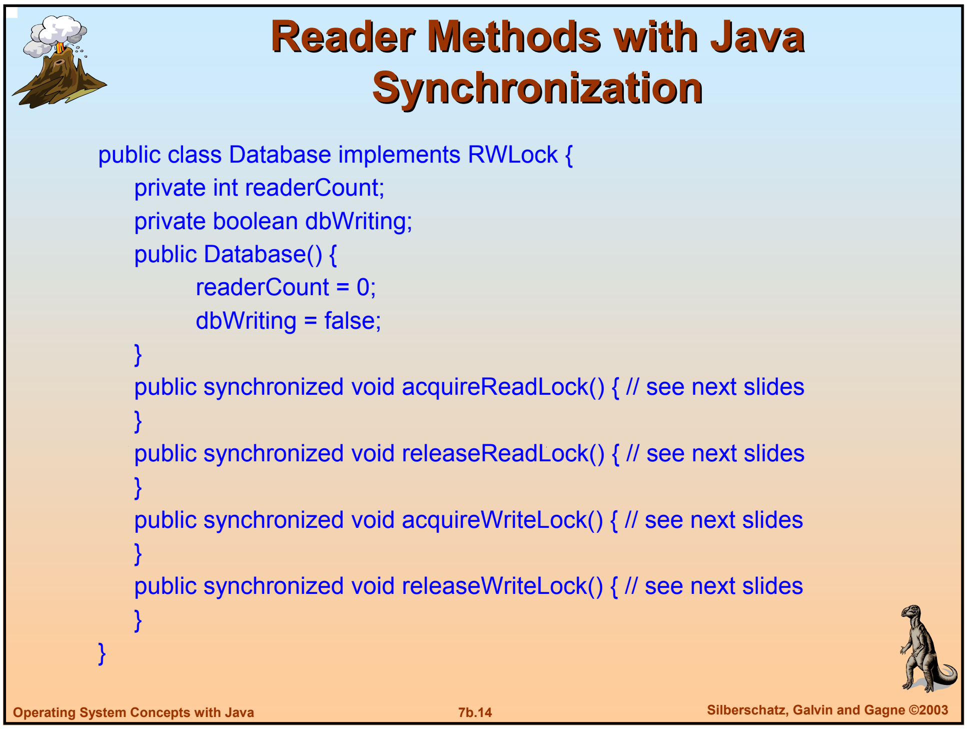
Complete Bounded Buffer using Java Synchronization

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER SIZE = 5;
    private int count, in, out;
    private Object[] buffer;
    public BoundedBuffer() { // buffer is initially empty
          count = 0;
          in = 0;
          out = 0;
          buffer = new Object[BUFFER SIZE];
    }
    public synchronized void insert(Object item) { // See previous slides
    }
    public synchronized Object remove() { // See previous slides
    }
}
```



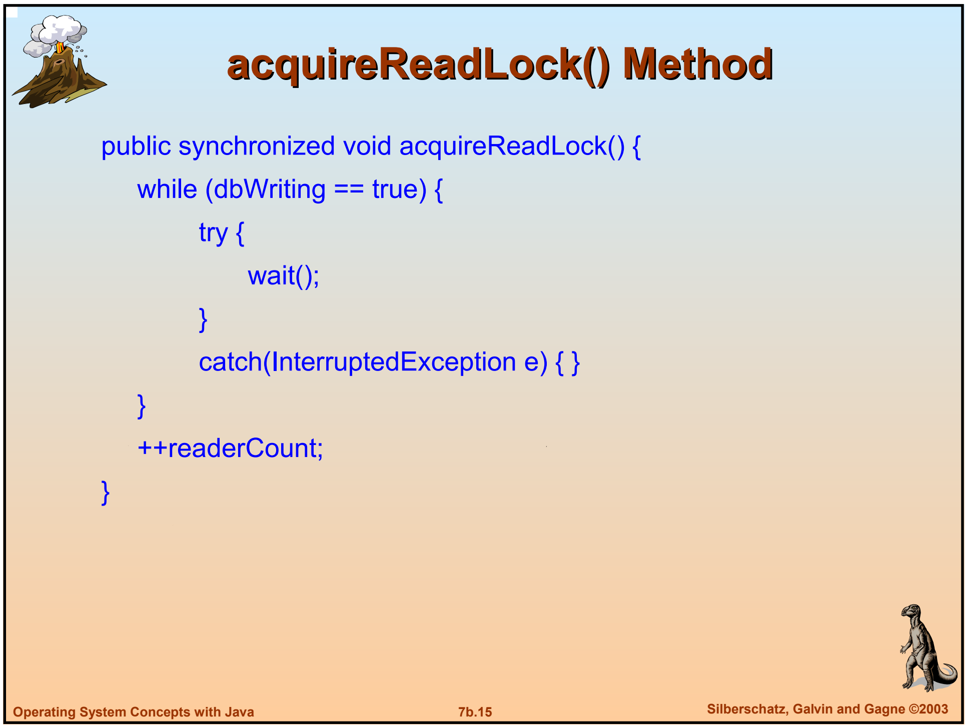
Multiple Notifications

- notify() selects an arbitrary thread from the wait set.
  *This may not be the thread that you want to be selected.
- Java does not allow you to specify the thread to be selected
- notifyAll() removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.
- notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set



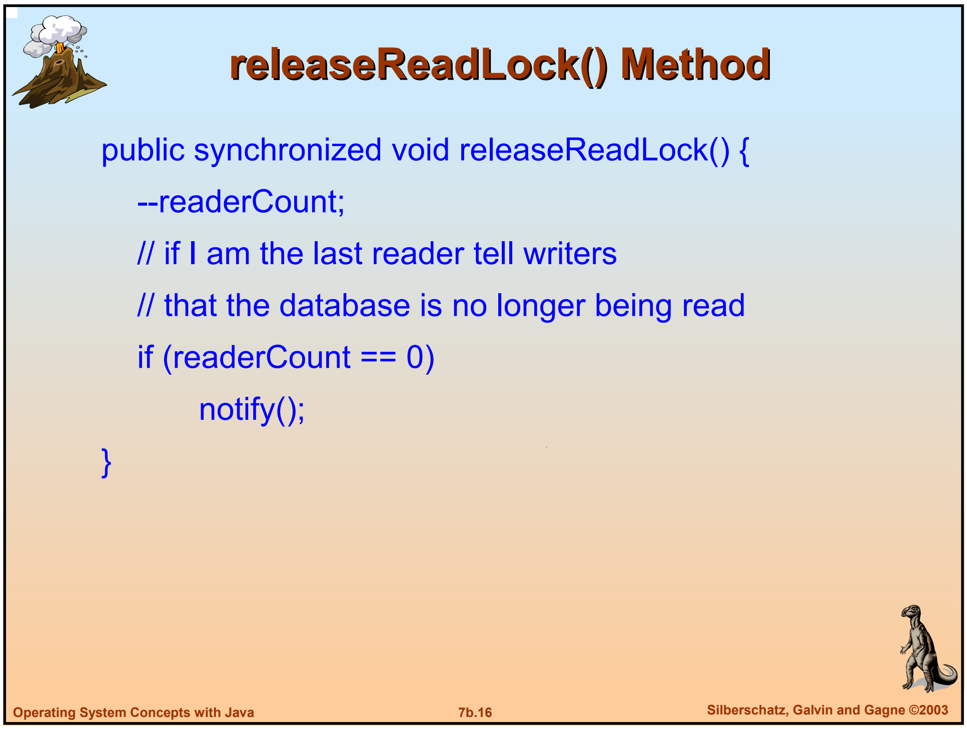
Reader Methods with Java Synchronization

```
public class Database implements RWLock {
    private int readerCount;
    private boolean dbWriting;
    public Database() {
           readerCount = 0;
           dbWriting = false;
    }
    public synchronized void acquireReadLock() { // see next slides
    }
    public synchronized void releaseReadLock() { // see next slides
    }
    public synchronized void acquireWriteLock() { // see next slides
    }
    public synchronized void releaseWriteLock() { // see next slides
    }
}
```



acquireReadLock() Method

```
public synchronized void acquireReadLock() {
   while (dbWriting == true) {
         try {
              wait();
         }
         catch(InterruptedException e) { }
   }
   ++readerCount;
}
```



releaseReadLock() Method

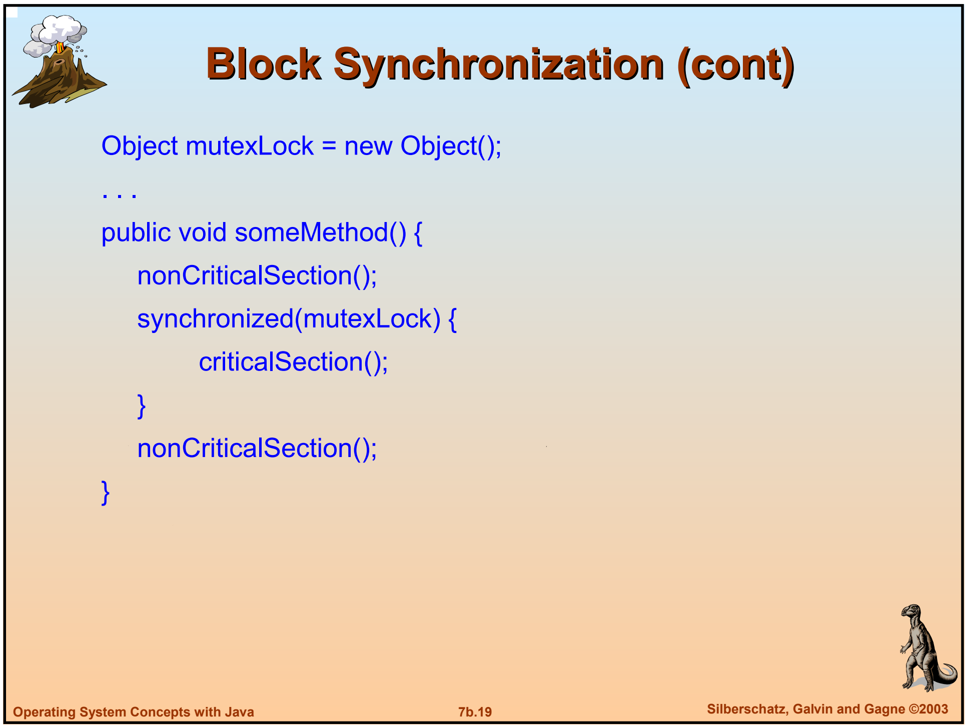
```
public synchronized void releaseReadLock() {
   --readerCount;
   // if I am the last reader tell writers
   // that the database is no longer being read
   if (readerCount == 0)
         notify();
}
```

Writer Methods

```
public synchronized void acquireWriteLock() {
    while (readerCount > 0 || dbWriting == true) {
          try {
               wait();
          }
          catch(InterruptedException e) { }
    }
    // once there are either no readers or writers
    // indicate that the database is being written
    dbWriting = true;
}
public synchronized void releaseWriteLock() {
    dbWriting = false;
    notifyAll();
}
```

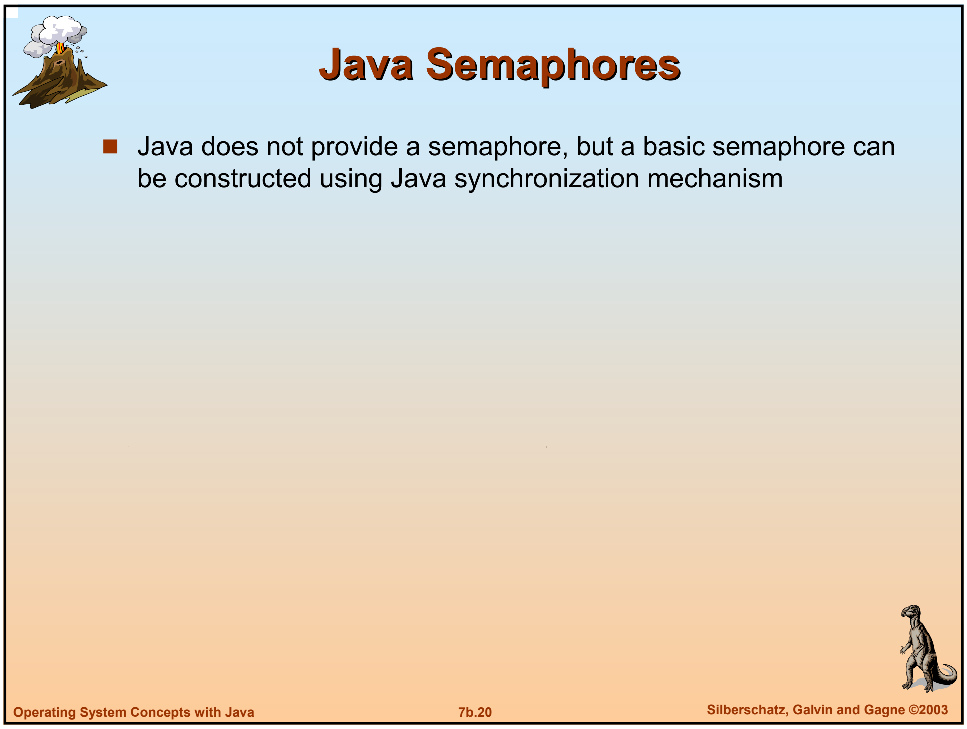
Block Synchronization

- Scope of lock is time between lock acquire and release
- Blocks of code – rather than entire methods – may be declared as **synchronized**
- This yields a lock scope that is typically smaller than a synchronized method



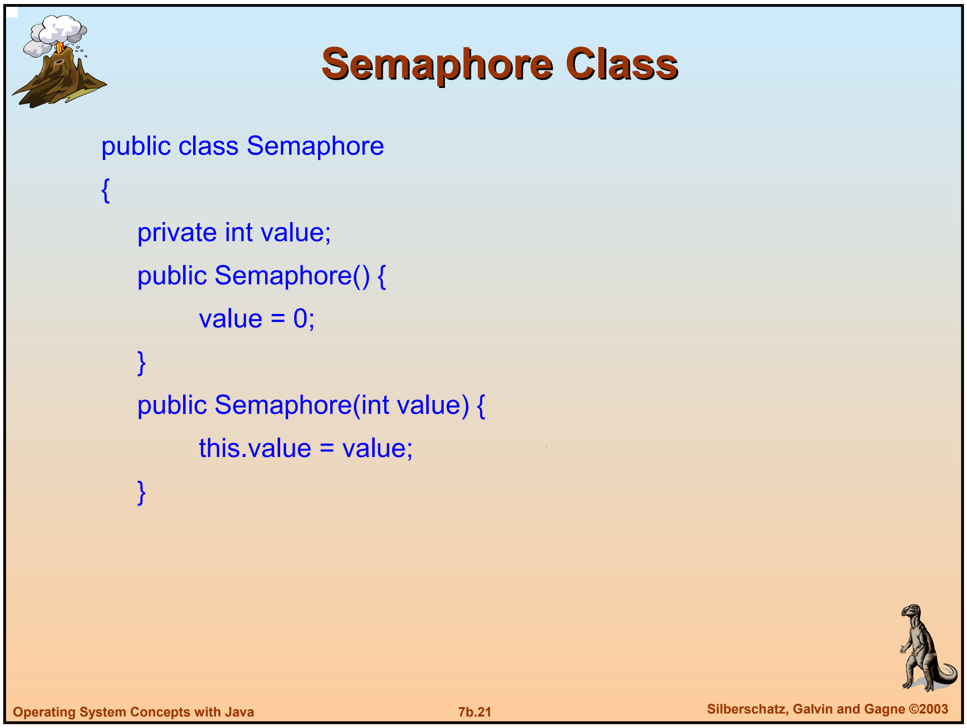
Block Synchronization (cont)

```
Object mutexLock = new Object();
. . .
public void someMethod() {
    nonCriticalSection();
    synchronized(mutexLock) {
          criticalSection();
    }
    nonCriticalSection();
}
```



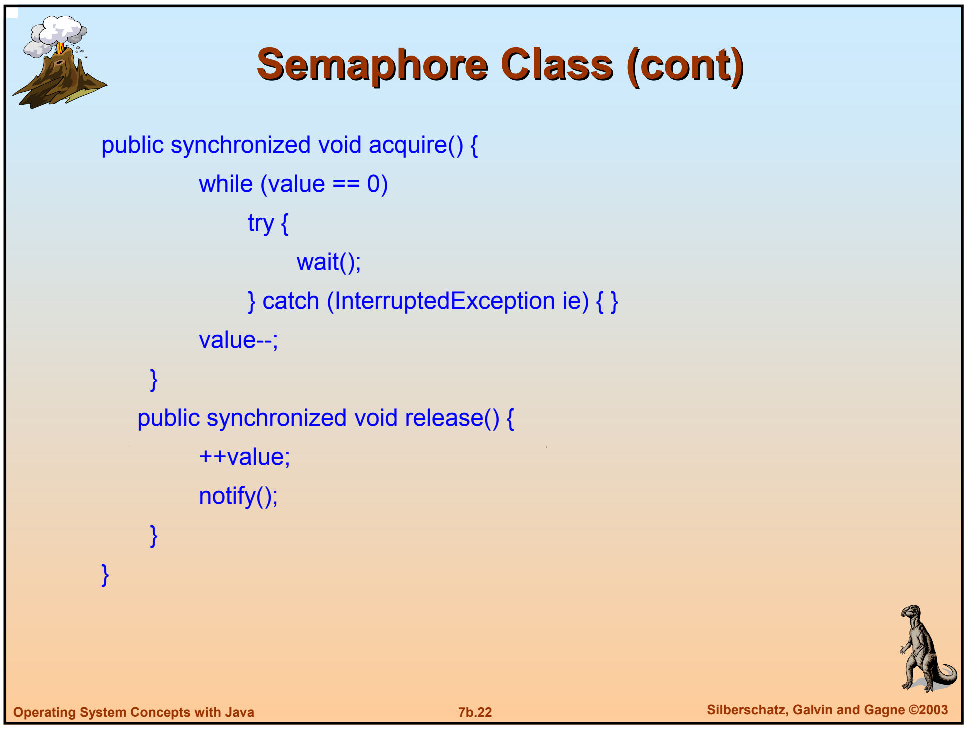
Java Semaphores

- Java does not provide a semaphore, but a basic semaphore can be constructed using Java synchronization mechanism



Semaphore Class

```
public class Semaphore
{
   private int value;
   public Semaphore() {
        value = 0;
   }
   public Semaphore(int value) {
        this.value = value;
   }
```



Semaphore Class (cont)

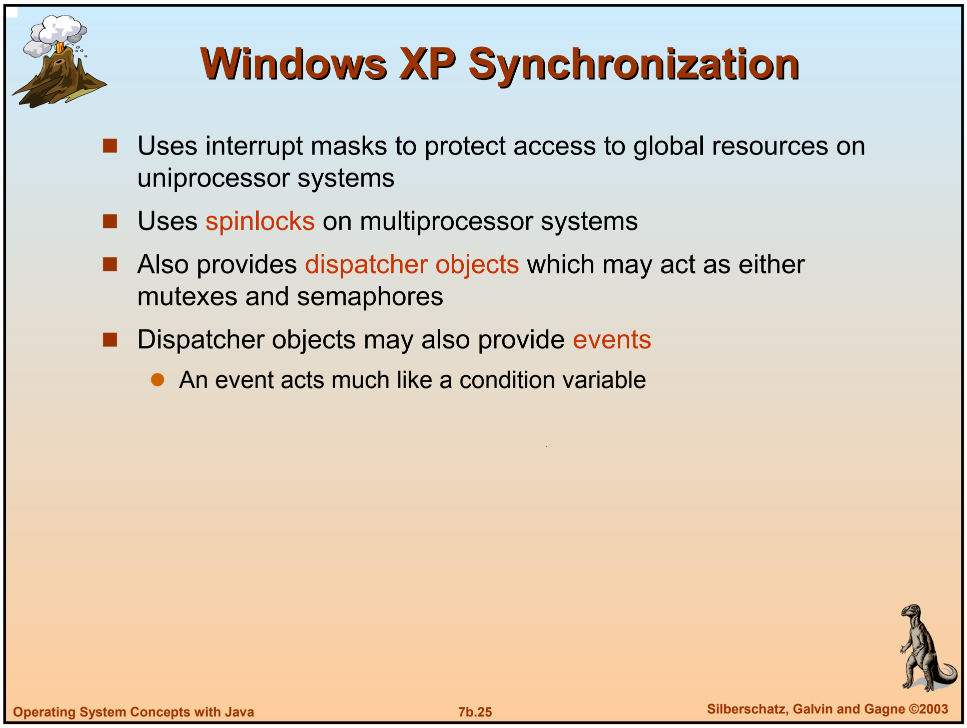
```
public synchronized void acquire() {
        while (value == 0)
            try {
                wait();
            } catch (InterruptedException ie) { }
        value--;
    }
   public synchronized void release() {
        ++value;
        notify();
    }
}
```

Syncronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

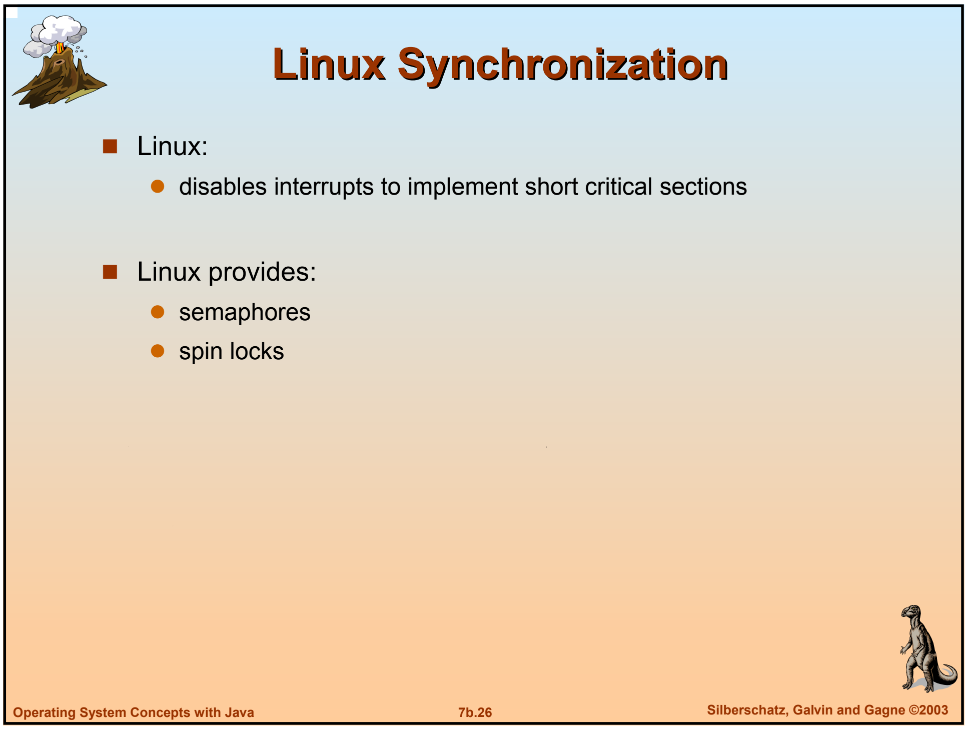
Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock



Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable



Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks