COMP 322: Fundamentals of Parallel Programming

Lecture 25: Java synchronized statement (cont)

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

One possible solution to Worksheet #24

1) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using start() and join() operations.

```
1.  // Start of thread t0 (main program)
2.  sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields
3.  // Compute sum1 (lower half) and sum2 (upper half) in parallel
4.  final int len = X.length;
5.  Thread t1 = new Thread(() -> {
6.              for(int i=0 ; i < len/2 ; i++) sum1+=X[i];});
7.  t1.start();
8.  Thread t2 = new Thread(() -> {
9.              for(int i=len/2 ; i < len ; i++) sum2+=X[i];});
10. t2.start();
11. int sum = sum1 + sum2; //data race between t0 & t1, and t0 & t2
12. t1.join(); t2.join();
```


One possible solution to Worksheet #24 (contd)

2) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using synchronized statements.

```
1.  // Start of thread t0 (main program)
2.  sum = 0; // static int field
3.  Object a = new ... ;
4.  Object b = new ... ;
5.  Thread t1 = new Thread(() ->
6.                  { synchronized(a) { sum++; } });
7.  Thread t2 = new Thread(() ->
8.                  { synchronized(b) { sum++; } });
9.  t1.start();
10. t2.start(); // data race between t1 & t2
11. t1.join(); t2.join();
```

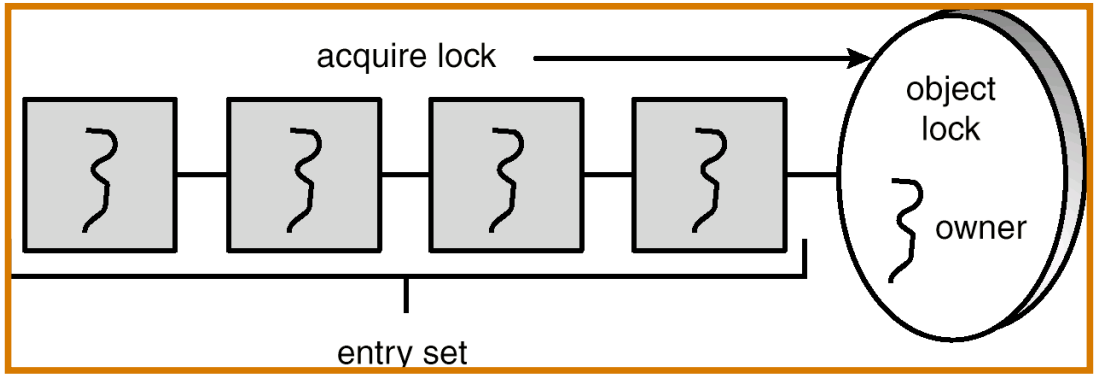

Monitors

- One definition of monitor is a thread-safe class, object, or module that uses wrapped mutual exclusion in order to safely allow access to a method or variable by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion: At each point in time, at most one thread may be executing any of its methods. Using a condition variable(s), it can also provide the ability for threads to wait on a certain condition (thus using the above definition of a "monitor"). For the rest of this article, this sense of "monitor" will be referred to as a "thread-safe object/ class/module".

- Source: https://en.wikipedia.org/wiki/Monitor_(synchronization)

Implementation of Java synchronized statements/methods

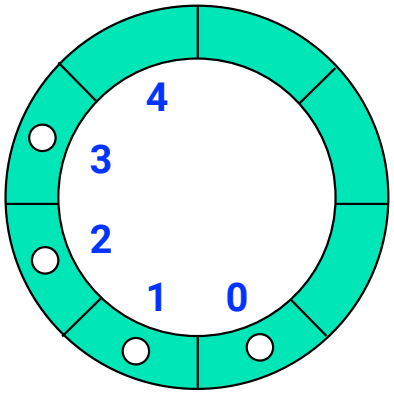
- Every object has an associated lock
- "synchronized" is translated to matching monitorenter and monitorexit bytecode instructions for the Java virtual machine
  - monitorenter requests "ownership" of the object's lock
  - monitorexit releases "ownership" of the object's lock
- If a thread performing monitorenter does not gain ownership of the lock (because another thread already owns it), it is placed in an unordered "entry set" for the object's lock



What if you want to wait for shared state to satisfy a desired property? (Circular Bounded Buffer Example)

```
1. public synchronized void insert(Object item) { // producer
2.     // TODO: wait till count < BUFFER SIZE
3.     ++count;
4.     buffer[in] = item;
5.     in = (in + 1) % BUFFER SIZE;
6.     // TODO: notify consumers
7. }
8.
9. public synchronized Object remove() { // consumer
10.    Object item;
11.    // TODO: wait till count > 0
12.    --count;
13.    item = buffer[out];
14.    out = (out + 1) % BUFFER SIZE;
15.    // TODO: notify producers
16.    return item;
17. }
```

count=4
out=0
in=4



The Java wait() Method

- A thread can perform a wait() method on an object that it owns:
  1. the thread releases the object lock
  2. thread state is set to blocked
  3. thread is placed in the wait set
- Causes thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- Since interrupts and spurious wake-ups are possible, this method should always be used in a loop e.g.,

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait();
    ... // Perform action appropriate to condition
}
```

- Java's wait-notify is related to "condition variables" in POSIX threads

Monitors – a Diagrammatic summary

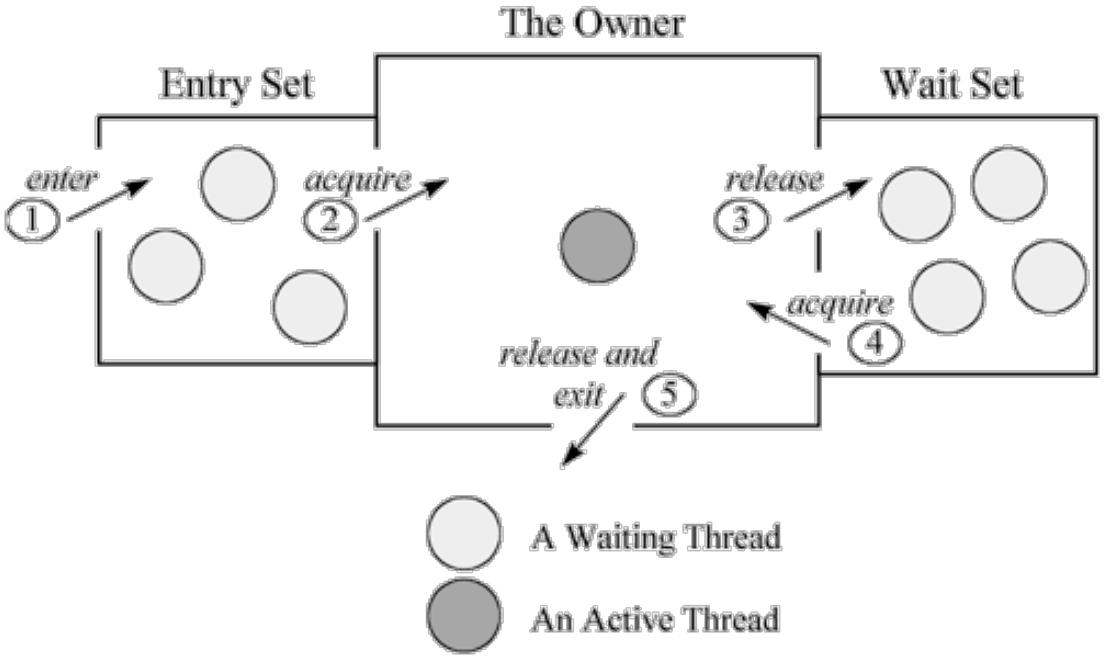
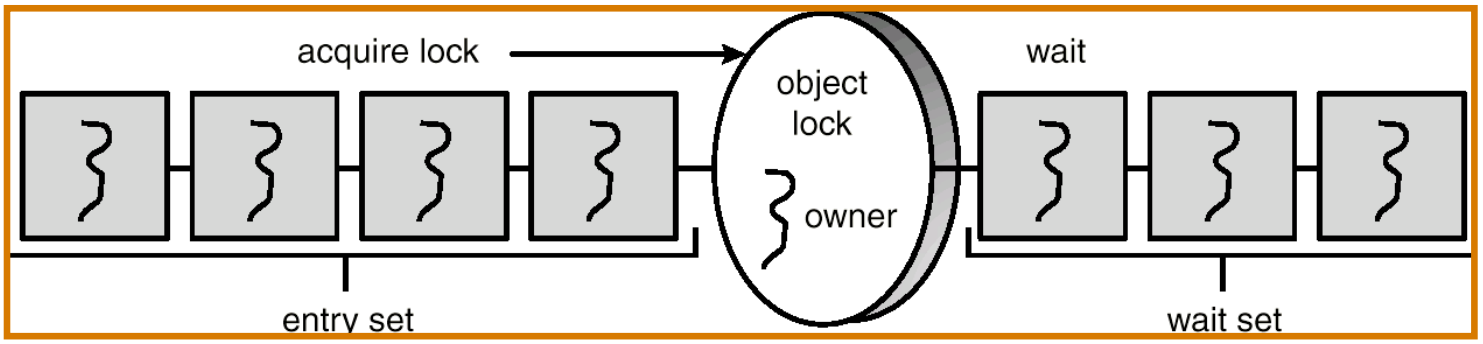


Figure 20-1. A Java monitor.

Figure source: http://www.artima.com/insidejvm/ed2/images/fig20-1.gif

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

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

insert() & remove() with wait/notify methods for Circular Bounded Buffer

```
1. public synchronized void insert(Object item) {
2.     while (count == BUFFER SIZE) wait();
3.     ++count;
4.     buffer[in] = item;
5.     in = (in + 1) % BUFFER SIZE;
6.     notify();
7. }
8.
9. public synchronized Object remove() {
10.     Object item;
11.     while (count == 0) wait();
12.     --count;
13.     item = buffer[out];
14.     out = (out + 1) % BUFFER SIZE;
15.     notify();
16.     return item;
17. }
```


Complete Bounded Buffer class using Java Synchronization

```
1. public class BoundedBuffer extends Buffer
2. {
3.     private static final int BUFFER SIZE = 5;
4.     private int count, in, out;
5.     private Object[] buffer;
6.     public BoundedBuffer() { // create empty buffer
7.        count = 0; in = 0; out = 0;
8.        buffer = new Object[BUFFER SIZE];
9.     }
10.     public synchronized void insert(Object item) {
11.       // See previous slide
12.     }
13.     public synchronized Object remove() {
14.       // See previous slide
15.     }
16. }
```