COMP 322: Fundamentals of Parallel Programming

Lecture 27: Java synchronized statement (contd), wait/notify operations

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Unit 7.1: Java Threads (Recap)

- **Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program's main() method.**
- **Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.**

```
1  public class Thread extends Object implements Runnable {
2    Thread() { ... } // Creates a new Thread
3    Thread(Runnable r) { ... } // Creates a new Thread with Runnable object r
4    void  run() { ... } //
5     // Case 1: If this thread was
6     //          then that object's run method
7     // Case 2: If this class is subclassed, t
8     //          in the subclass is called
9    void start() { ... } // Causes this thread to
10   void join() { ... } // Wait for this thread to die
11   void  join(long m) // Wait at most m milliseconds for thread to die
12   static Thread currentThread() // Returns currently executing thread
13   . . .
14 }
```

A lambda can be passed as a Runnable

Listing 3: java.lang.Thread class

Solution to Worksheet #26: Java Threads

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();
```


Solution to Worksheet #26: Java Threads (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(() -> { synchronized(a) { sum++; } });
6.  Thread t2 = new Thread(() -> { synchronized(b) { sum++; } });
1.  t1.start();
7.  t2.start(); // data race between t1 & t2
8.  t1.join(); t2.join();
```


Unit 7.2: Objects and Locks in Java --- synchronized statements and methods

- Every Java object has an associated *lock* acquired via:
  - `synchronized` statements
    - synchronized(foo) { // acquire foo's lock
      // execute code while holding foo's lock
      } // release foo's lock
  - `synchronized` methods
    - public synchronized void op1() { // acquire 'this' lock
      // execute method while holding 'this' lock
      } // release 'this' lock
- Java language does not enforce any relationship between object used for locking and objects accessed in isolated code
  - If same object is used for locking and data access, then the object behaves like a monitor
- Locking and unlocking are automatic
  - Locks are released when a synchronized block exits
    - By normal means: end of block reached, return, break
    - When an exception is thrown and not caught

Deadlock example with Java synchronized statement

- The code below can deadlock if `leftHand()` and `rightHand()` are called concurrently from different threads
  - Because the locks are not acquired in the same order

```
public class ObviousDeadlock {
    . . .
    public void leftHand() {
        synchronized(lock1) {
            synchronized(lock2) {
                for (int i=0; i<10000; i++)
                    sum += random.nextInt(100);
            }
        }
    }
    public void rightHand() {
        synchronized(lock2) {
            synchronized(lock1) {
                for (int i=0; i<10000; i++)
                    sum += random.nextInt(100);
            }
        }
    }
}
```


Object-based isolation in HJ does not deadlock

```
public class NoDeadlock2 {
        public void transferFunds(Account from,
                                  Account to,
                                  int amount) {
            isolated (from, to) {
                    from.subtractFromBalance(amount);
                    to.addToBalance(amount);
                } } } }
```

- HJ's implementation guarantees that object-based isolation is deadlock-free
- However, HJ does not permit an inner isolated statement to add a new object e.g., the following code is not permitted in HJ, but the equivalent synchronized version is permitted in Java

Not permitted in HJ (if from != to)

```
isolated (from) {
  ...
  isolated (to) { . . .}
}
```

Permitted in Java

```
synchronized (from) {
  ...
  synchronized(to) { . . .}
}
```


Deadlock avoidance in HJ with object-based isolation

- HJ implementation ensures that all locks are acquired by the runtime in the same order
- ==> no deadlock

```
public class NoDeadlock1 {
    . . .
    public void leftHand() {
        isolated(lock1, lock2) {
                for (int i=0; i<10000; i++)
                    sum += random.nextInt(100);

        }
    }
    public void rightHand() {
        isolated(lock2,lock1) {
                for (int i=0; i<10000; i++)
                    sum += random.nextInt(100);
            }
        }
    }
}
```

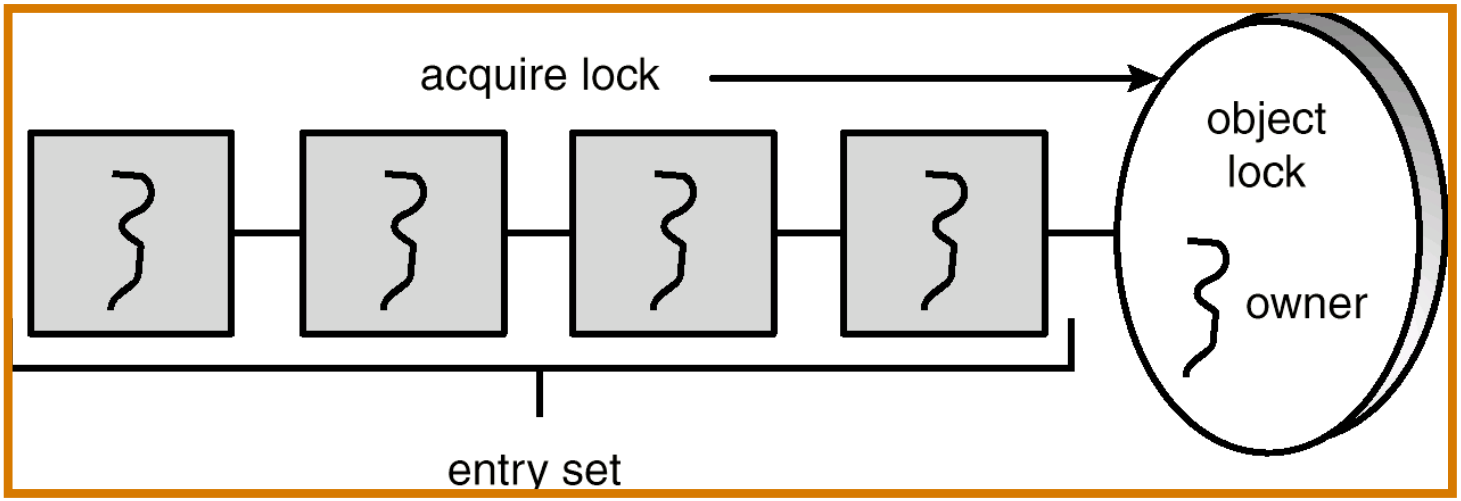

Java's Object Locks are Reentrant

- Locks are granted on a per-thread basis
  - Called reentrant or recursive locks
  - Promotes object-oriented concurrent code
- A synchronized block means execution of this code requires the current thread to hold this lock
  - If it does — fine
  - If it doesn't — then acquire the lock
- Reentrancy means that recursive methods, invocation of super methods, or local callbacks, don't deadlock

```
public class Widget {
    public synchronized void doSomething() { ... }
}
public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        Logger.log(this + ": calling doSomething()");
        super.doSomething();  // Doesn't deadlock!
    }
}
```


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



Monitors – a Diagrammatic summary

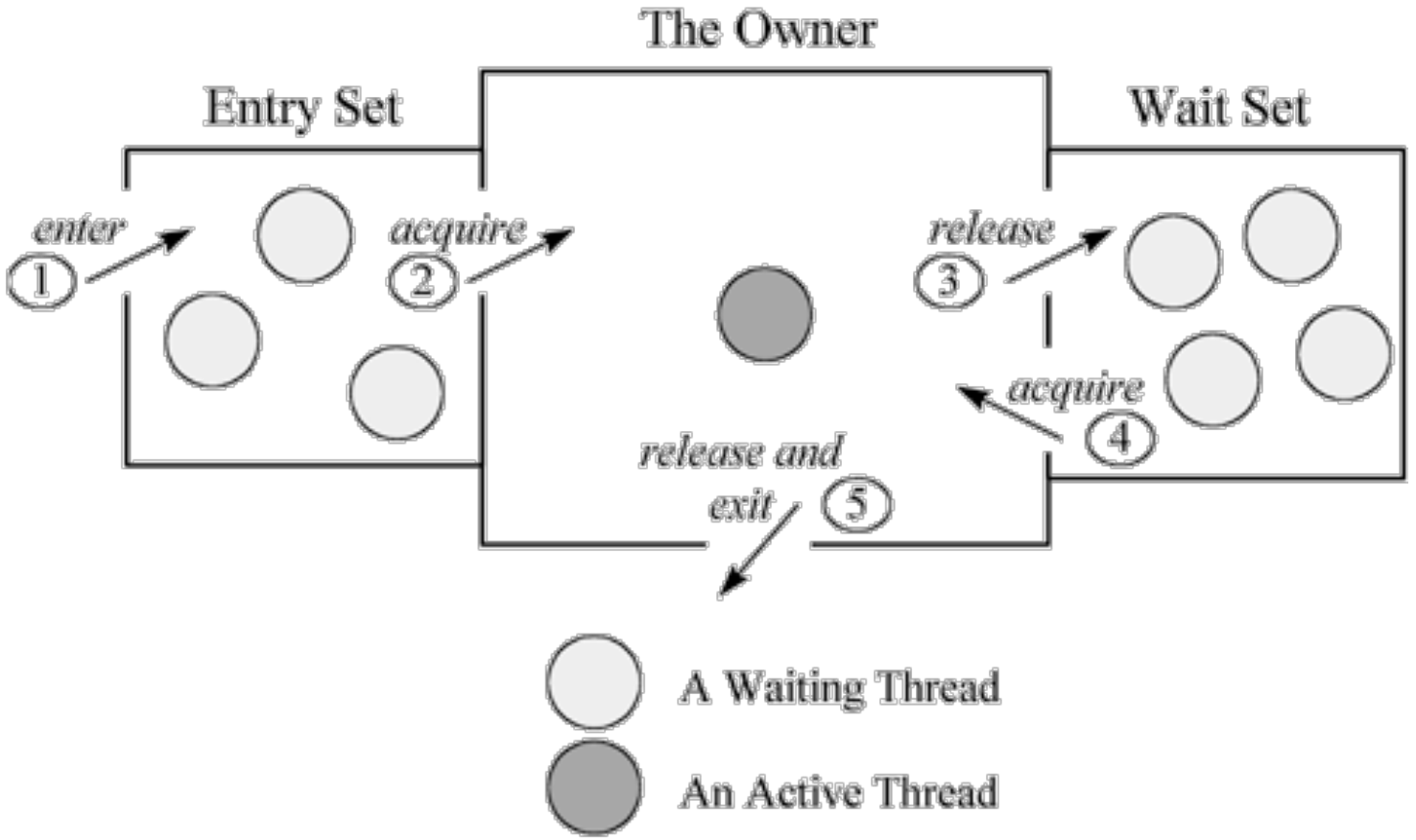


Figure 20-1. A Java monitor.

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

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

```
public synchronized void insert(Object item) { // producer
    // TODO: wait till count < BUFFER SIZE
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
    // TODO: notify consumers that an insert has been performed
}

public synchronized Object remove() { // consumer
    Object item;
  // TODO: wait till count > 0
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    // TODO: notify producers that a remove() has been performed
    return item;
}
```

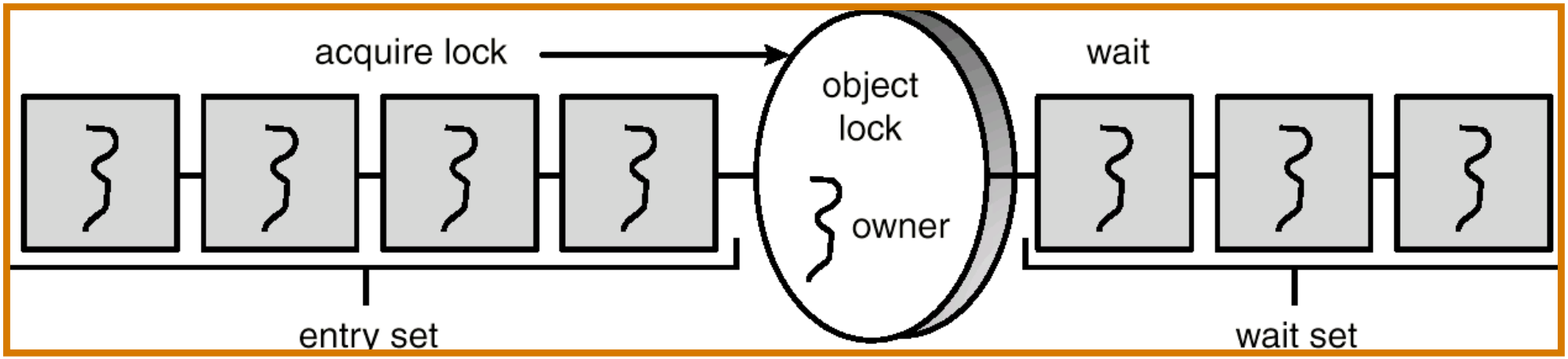

The Java wait() Method

- 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

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() 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
    }
}
```