

---

# ***SYSC 3303 Real-Time Concurrent Systems***

## **Synchronizing Java Threads, Continued**

- Copyright © 2001-2004 D.L. Bailey, 2005-2012 L.S. Marshall Systems and Computer Engineering, Carleton University
- revised July 12<sup>th</sup>, 2012

---

## ***Synchronized Statements***

- A statement can be turned into a critical section by placing it in a *synchronized statement*
- Syntax:  

```
synchronized (object-reference)  
statement
```

  - statement can be a compound statement
- When a thread executes a `synchronized` statement, it first attempts to obtain the lock for the specified object
- If the thread obtains the lock, it starts executing the statement

---

## ***Synchronized Statements***

- If the thread cannot obtain the lock (because another thread holds the lock), the thread *blocks* (it is no longer eligible to run)
- When a thread finishes executing a `synchronized` statement, it releases the object's lock
- Here is the `Box` class, rewritten to use synchronized statements
  - this version is completely equivalent to the version that has `synchronized` methods

---

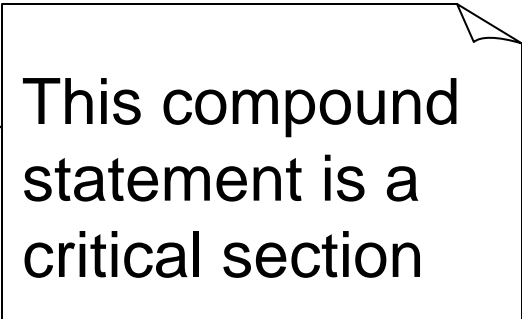
## ***Box with Synchronized Statements***

```
public class Box {  
  
    /**  
     * The object stored in this Box.  
     */  
    private Object contents = null;  
  
    /**  
     * State of the box.  
     */  
    private boolean empty = true;  
}
```

---

## ***Box with Synchronized Statements***

```
public void put(Object obj) {  
    synchronized (this) {  
        while (!empty) {  
            wait();  
        }  
        contents = obj;  
        empty = false;  
        notifyAll();  
    }  
}
```



This compound statement is a critical section

---

## ***Box with Synchronized Statements***

```
public Object get() {  
    synchronized (this) {  
        while (empty) {  
            wait();  
        }  
        Object obj = contents;  
        empty = true;  
        notifyAll();  
        return obj;  
    }  
}
```

A critical section

---

## ***Example: A Work Queue***

- `WorkQueue` is a class that allows clients to enqueue work items for processing by a background thread
- The constructor creates and starts a background thread (an instance of private class `WorkerThread`) that removes items from the queue and processes them by invoking `processItem()`
- `WorkQueue` is an abstract class - the intent is that users will develop a concrete subclass that provides a concrete implementation of `processItem()` that is appropriate for the work items

---

## ***Example: A Work Queue***

```
public abstract class WorkQueue
{
    private List queue = new LinkedList();

    protected WorkQueue() {
        new WorkerThread().start();
    }

    public final void enqueue(Object workItem) {
        synchronized (queue) {
            queue.add(workItem);
            queue.notify();
        }
    }
}
```



---

## ***Example: A Work Queue***

```
protected abstract void
processItem(Object workItem);

private class WorkerThread extends Thread
{
    public void run() {
        Object workItem = null;
        while (true) {
            synchronized (queue) {
                try {
                    while (queue.isEmpty())
                        queue.wait();
                } catch (InterruptedException e) {
                    return;
                }
            }
        }
    }
}
```

---

---

## ***Example: A Work Queue***

```
        workItem = queue.removeFirst();  
    } // end of critical section  
    processItem(workItem);  
}  
}  
}  
}
```

---

## ***Why Use Synchronized Statements?***

- This example illustrates two uses for synchronized statements:
  - external synchronization on an object that is not thread-safe
  - increasing concurrency by reducing the length of a critical section

---

## ***External Synchronization***

- The queue is an instance of `LinkedList` (a class in the Java Collections Framework), which is not thread-safe
  - external synchronization is required to enforce mutual exclusion
  - threads use synchronized statements to obtain the queue's lock before invoking queue operations

---

## ***Length of Critical Sections***

- `processItem( )` does not need access to the shared queue, so it is invoked outside of the critical section (i.e., outside the synchronized statement)
- Because the `WorkerThread` relinquishes the queue's lock before invoking `processItem( )`, client threads can add items to the queue while the removed work item is handled by the `WorkerThread`, increasing concurrency

---

## ***Length of Critical Sections***

- To increase concurrency, do as little work as possible inside a critical section
- If a critical section contains a time-consuming activity, find a way to move the activity outside of the critical section
  - obviously, shared data must always be accessed in a critical section
- If a synchronized method contains a time-consuming activity, make it non-synchronized, and use a synchronized statement to provide the critical section within the method (with the activity moved outside of the critical section)

---

## ***Example: The Java Collections Framework***

- Most of the classes in the Java Collections Framework (e.g., `ArrayList`, `LinkedList`, `HashMap`, `WeakHashMap`, `TreeMap`, `HashSet`, `TreeSet`) are not thread-safe
- The `Collections` class provides static methods that return a thread-safe collection that is backed by an unsynchronized collection

---

## ***Example: The Java Collections Framework***

- Example: to create a thread-safe `Collection`:

```
Collection c =  
    Collections.synchronizedCollection(  
        new ArrayList());
```

- The argument is an instance of any class that implements the `Collection` interface
- example: to create a thread-safe `Map`:

```
Map map =  
    Collections.synchronizedMap(new TreeMap());
```

- The argument is an instance of any class that implements the `Map` interface



---

## ***Example: The Java Collections Framework***

- Messages that are sent to a synchronized collection are forwarded to the backing collection, but the synchronized class guarantees that access to the backing collection is serialized
- How does it do this?
- Let's look at the code for the object returned by `synchronizedCollection()`

---

## ***Example: The Java Collections Framework***

```
public static Collection
synchronizedCollection(Collection c) {
    return new SynchronizedCollection(c);
}

static class SynchronizedCollection
implements Collection, Serializable {
    Collection c;    // Backing Collection
    Object mutex;    // Object on which to synch

    SynchronizedCollection(Collection c) {
        this.c = c;
        mutex = this;
    }
}
```

---

## ***Example: The Java Collections Framework***

```
public int size() {  
    synchronized(mutex) {return c.size();}  
}  
  
public boolean isEmpty() {  
    synchronized(mutex) {return c.isEmpty();}  
}  
...  
  
public boolean contains(Object o) {  
    synchronized(mutex) {return c.contains(o);}  
}  
...
```

---

## ***Example: The Java Collections Framework***

```
public boolean add(Object o) {  
    synchronized(mutex) {return c.add(o);}  
}  
  
public boolean remove(Object o) {  
    synchronized(mutex) {return c.remove(o);}  
}  
    ...  
}
```

---

## ***Example: The Java Collections Framework***

- When a `SynchronizedCollection` is created, it saves a reference to the mutex object on which clients will synchronize
  - the statement `mutex = this;` in the constructor arranges for the `SynchronizedCollection` object itself to be the mutex object
    - other constructors (not shown) are passed a separate mutex object as an argument, and the reference to that object is saved in `mutex`

---

## ***Example: The Java Collections Framework***

- When a thread sends a message to an instance of `SynchronizedCollection`, it first obtains the mutex object's lock, then forwards the message to the backing collection; e.g.,

```
public boolean add(Object o) {  
    synchronized(mutex) {  
        return c.add(o);  
    }  
}
```

---

## Synchronized Statement in Static Method

- This example is taken from:  
[http://java.sun.com/docs/books/jls/first\\_edition/html/8.doc.html#55408](http://java.sun.com/docs/books/jls/first_edition/html/8.doc.html#55408)

- Class Test with synchronized methods:

```
class Test {  
    int count;  
    synchronized void bump() { count++; }  
    static int classCount;  
    static synchronized void classBump() { classCount++; }  
}
```

---

## Synchronized Statement in Static Method

- Class BumpTest with synchronized statements:

```
class BumpTest {  
    int count;  
    void bump() {  
        synchronized (this) {  
            count++;  
        }  
    }  
}
```

// continued



---

## Synchronized Statement in Static Method

```
static int classCount;
static void classBump() {
    try { synchronized (Class.forName("BumpTest")) {
        classCount++;
    }
    } catch (ClassNotFoundException e) { ... }
}
```

---

## And a Less Verbose classBump with Synchronized Statement

```
static void classBump() {  
    synchronized (BumpTest.class) {  
        classCount++;  
    }  
}  
}
```

---

## ***Documenting Thread Safety***

- Classes must document the extent to which their methods can be executed concurrently by multiple threads (ref: *Effective Java*, Item 52)
  - the API documentation for many (most?) Java classes does not indicate how its instances and static methods behave in the presence of concurrent threads
    - this is not unique to Java - the same problem is common to C++ class/C function libraries
  - the following slides describe one approach to categorizing levels of thread safety

---

## ***1. Immutable Classes***

- Instances of immutable classes cannot be changed after they are initialized
- Their methods can be safely invoked by concurrent threads
  - no thread synchronization required
  - examples: `String`, `Integer`

---

## ***2. Thread-Safe Classes***

- Instances of thread-safe class are mutable, but all methods contain sufficient *internal* synchronization that instances may be used concurrently without the need for *external* synchronization

---

### ***3. Conditionally Thread-Safe Classes***

- These classes are similar to thread-safe classes except that the classes (or associated classes) contain some methods that must be invoked in sequence without interference from other threads
- We must document which invocation sequences require external synchronization and which lock or locks must be acquired to prevent concurrent access

---

### ***3. Conditionally Thread-Safe Classes***

- Example: Methods defined in the synchronized collection wrapper classes (e.g., `SynchronizedCollection`, `SynchronizedList`, `SynchronizedSet`, etc.) guarantee mutually exclusive access to the backing collection (this is documented in the Java Platform API Specification)
- Recall that many of these classes have methods that return `Iterator` objects that allow clients to iterate through the elements in the collection

---

### ***3. Conditionally Thread-Safe Classes***

- The standard idiom for iterating over a collection is:

```
List list =  
    Collections.synchronizedList(  
        new ArrayList());  
  
...  
Iterator i = list.iterator();  
while (i.hasNext()) {  
    Object o = i.next();  
    ... // do something with o  
}
```



---

### ***3. Conditionally Thread-Safe Classes***

- Even though the synchronized collection is thread-safe, iterating over the collection this way is not guaranteed to work
  - there is no way for the `Iterator` object to lock the collection for the duration of the iteration
  - if one thread changes the collection while another thread is iterating over it, the behaviour of the `Iterator` methods is non-deterministic
- Clients must lock the entire synchronized collection prior to requesting the `Iterator` object, and must retain the lock until the iteration is finished

---

## ***3. Conditionally Thread-Safe Classes***

- Here is the correct way to perform thread-safe iteration over a thread-safe collection

```
List list =  
    Collections.synchronizedList(  
        new ArrayList());  
  
...  
synchronized (list) {  
    Iterator i = list.iterator();  
    while (i.hasNext()) {  
        Object o = i.next();  
        ... // do something with o  
    }  
}
```

---

## ***4. Thread-Compatible Classes***

- Instances of thread-compatible classes can safely be used concurrently by surrounding each method invocation (and in some cases, each sequence of method invocations) by external synchronization
- Examples: the collection classes added in Java 2 SDK Version 1.2 and subsequent releases; e.g., `ArrayList` and `HashMap`
- We've seen how the synchronized collection wrapper classes perform the required external synchronization

---

## ***5. Thread-Hostile Classes***

- Thread-hostile classes cannot safely be used concurrently by multiple threads even if all method invocations are surrounded by external synchronization
- Typically, these methods modify class (`static`) variables that affect other threads
- There are very few thread-hostile classes or methods in the Java platform libraries

---

## ***wait( ) With Timeout***

- The version of `wait( )` that we've used until now causes the invoking `Thread` to block on an object until another `Thread` invokes `notify( )` or `notifyAll( )` for the object
- Class `Object` defines two other `wait( )` methods
- Both methods block the `Thread` until
  - another `Thread` invokes `notify( )` or `notifyAll( )` for the object, or
  - a specified amount of time has elapsed

---

## ***wait( ) With Timeout***

- `public final void wait(long timeout)`  
throws `InterruptedException`
- Parameter `timeout` specifies the maximum time to wait in milliseconds
- If `timeout` is 0 the thread waits until it is notified; i.e.; `wait(0)` is equivalent to `wait( )`
- If `timeout` is negative this method throws an `IllegalArgumentException`

---

## ***wait( ) With Timeout***

- `public final void wait(long timeout,  
int nanos)`  
throws `InterruptedException`
- Parameter `timeout` specifies the time to wait in milliseconds
- Parameter `nanos` specifies additional time to wait, in nanoseconds
  - this value must be in the range 0-999999
- The maximum waiting time, in ns, is  
`1,000,000 * timeout + nanos`

---

## ***wait( ) With Timeout***

- If `timeout` and `nanos` are 0 the thread waits until it is notified; i.e.; `wait(0, 0)` is equivalent to `wait()`
- If `timeout` is negative or `nanos` is not in the range 0 to 999999 this method throws an `IllegalArgumentException`



---

## ***wait( ) With Timeout***

- How does a Thread determine why it returned from `wait(long)` or `wait(long, int)`
  - was it notified or did it time out?
  - `wait( )` does not return a special value or throw an exception if it times out
- The Thread that invokes `notify( )/notifyAll( )` will typically set a flag or condition variable in the shared object prior to notifying the waiting thread(s)
- Upon returning from `wait( )`, a thread checks this variable - if it wasn't notified, it must have timed out

---

## ***How Can One Thread Stop Another?***

- Never use `stop( )` (it's deprecated)
- Use this idiom:
  - arrange for the thread to poll a private flag variable that indicates whether the thread should return from its `run( )` method
  - the thread's class provides methods to set and get this variable - both methods must be synchronized

---

## ***A Stoppable Thread***

```
public class StoppableThread extends Thread
{
    private boolean stopRequested = false;

    public void run() {
        boolean done = false;
        ...
        while (!stopRequested() && !done) {
            ... // do something
        }
    }
}
```

---

## ***A Stoppable Thread***

```
public synchronized void requestStop() {  
    stopRequested = true;  
}  
  
private synchronized boolean stopRequested() {  
    return stopRequested;  
}  
}
```

---

## ***A Stoppable Thread***

- A thread sends a `StoppableThread` the `requestStop()` message to ask it to finish whatever it is doing and return from its `run()` method (which will cause the `StoppableThread` to die)

```
StoppableThread t = new StoppableThread();  
t.start();  
...  
t.requestStop();
```

---

## ***A Stoppable Thread***

- The drawback to this approach is that if the stoppable thread is blocked (after invoking `wait()` or `sleep()`) when `requestStop()` is invoked, there can be a considerable delay before the thread is reenabled and notices that it has been asked to stop
- Fortunately, there is a way to forcibly unblock blocked threads

---

## ***Interrupt Methods***

- Every Thread maintains an *interrupted status*
- Class Thread provides three methods to set/check this status
- `public void interrupt()` interrupts the Thread that receives the `interrupt()` message
- If the thread is running or ready-to-run, the method simply sets the thread's interrupted status to true
  - despite the method's name, this method does not trigger the execution of the Java equivalent of an interrupt service routine, analogous to the ones you saw in SYSC 2003

---

## ***Interrupt Methods***

- Sending `interrupt()` to a Thread that is blocked in a wait set or is sleeping will cause the thread to unblock and return early from `wait()` and `sleep()`, throwing an `InterruptedException`
- `InterruptedException` is a checked exception, which is why we've always invoked `wait()` and `sleep()` from within a try/catch block



---

## ***Interrupt Methods***

- `public static boolean interrupted()`  
returns `true` if the current thread has been interrupted, `false` otherwise
- It clears the thread's interrupted status
- Example: suppose a `Thread` has been interrupted
- `Thread.interrupted()`; returns `true`
- A second invocation of `Thread.interrupted()` returns `false`, unless the current thread is interrupted again, after the first invocation cleared the thread's interrupt status and before the second invocation examined it

---

## ***Interrupt Methods***

- `public boolean isInterrupted()` returns `true` if this thread has been interrupted; `false` otherwise
- Unlike `interrupted()`, this method does not clear the thread's interrupted status

---

## ***Why Are The Interrupt Methods Useful?***

- The following version of the `WorkQueue` class shows how we can interrupt a thread to inform it that it should terminate

```
public abstract class WorkQueue
{
    private List queue = new LinkedList();
    private Thread worker;

    protected WorkQueue() {
        worker = new WorkerThread();
        worker.start();
    }
}
```

---

## ***Example: A Stoppable Work Queue***

```
public final void enqueue(Object workItem) {
    synchronized (queue) {
        queue.add(workItem);
        queue.notify();
    }
}

public final void stop() {
    worker.interrupt();
}
}
```

---

## ***Example: A Stoppable Work Queue***

```
protected abstract void
processItem(Object workItem);

private class WorkerThread extends Thread
{
    public void run() {
        Object workItem = null;
        while (!isInterrupted()) {
            synchronized (queue) {
                try {
                    while (queue.isEmpty())
                        queue.wait();
                } catch (InterruptedException e) {
                    return;
                }
            }
        }
    }
}
```

---

## ***Example: A Stoppable Work Queue***

```
        workItem = queue.removeFirst();
    } // end of critical section
    processItem(workItem);
} // end while
}
}
}
```

---

## ***Stopping A Thread***

- If the thread is interrupted while it is waiting, `wait()` throws an `InterruptedException`
- `run()` catches this exception and executes `return;` causing the thread to terminate
- If the thread is interrupted while it is removing a work item from the queue or processing the work item, its interrupted status is set `true`
- The thread checks this by invoking `isInterrupted()` after processing the work item, and exits `run()` if this method returns `true`