

# COMP 322: Fundamentals of Parallel Programming

## Lecture 25: Atomics, Java Synchronized Statements

Mack Joyner  
[mjoyner@rice.edu](mailto:mjoyner@rice.edu)

<http://comp322.rice.edu>



# How to enforce mutual exclusion?

---

- The predominant approach to ensure mutual exclusion proposed many years ago is to enclose the code region in a critical section.
  - “In concurrent programming a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore.”
  - Source: [http://en.wikipedia.org/wiki/Critical\\_section](http://en.wikipedia.org/wiki/Critical_section)



# java.util.concurrent.atomic.AtomicInteger

---

- Constructors
  - [new AtomicInteger\(\)](#)
    - Creates a new AtomicInteger with initial value 0
  - [new AtomicInteger\(int initialValue\)](#)
    - Creates a new AtomicInteger with the given initial value
- Selected methods
  - [int addAndGet\(int delta\)](#)
    - Atomically adds delta to the current value of the atomic variable, and returns the new value
  - [int getAndAdd\(int delta\)](#)
    - Atomically returns the current value of the atomic variable, and adds delta to the current value
- Similar interfaces available for LongInteger



# java.util.concurrent.AtomicInteger methods and their equivalent isolated constructs (pseudocode)

j.u.c.atomic Class and Constructors	j.u.c.atomic Methods	Equivalent HJ isolated statements
<b>AtomicInteger</b>	int j = v.get(); v.set(newVal);	int j; <b>isolated</b> (v) j = v.val; <b>isolated</b> (v) v.val = newVal;
<b>AtomicInteger()</b> // init = 0	int j = v.getAndSet(newVal); int j = v.addAndGet(delta); int j = v.getAndAdd(delta);	int j; <b>isolated</b> (v) { j = v.val; v.val = newVal; } <b>isolated</b> (v) { v.val += delta; j = v.val; } <b>isolated</b> (v) { j = v.val; v.val += delta; }
<b>AtomicInteger(init)</b>	boolean b = v.compareAndSet (expect,update);	boolean b; <b>isolated</b> (v) if (v.val==expect) {v.val=update; b=true;} else b = false;

Methods in java.util.concurrent.AtomicInteger class and their equivalent HJ isolated statements.

Variable v refers to an AtomicInteger object in column 2 and to a standard non-atomic Java object in column 3. val refers to a field of type int.



# Work-Sharing Pattern using AtomicInteger

```
1. import java.util.concurrent.atomic.AtomicInteger;  
2. ...  
3. String[] X = ... ; int numTasks = ...; int j;  
4. int[] taskId = new int[X.length];  
5. ...  
6. finish(() -> {  
7.     for (int i=0; i<numTasks; i++ )  
8.         async(() -> {  
9.             do {  
10.                 j = j + 1;  
11.                 // check if at end of X  
12.                 if (j >= X.length) break;  
13.                 taskId[j] = i; // Task i processes string X[j]  
14.             ...  
15.         } while (true);  
16.     });  
17.}); // finish-for-async
```



# Work-Sharing Pattern using AtomicInteger

```
1. import java.util.concurrent.atomic.AtomicInteger;  
2. ...  
3. String[] X = ... ; int numTasks = ...; int j;  
4. int[] taskId = new int[X.length];  
5. AtomicInteger a = new AtomicInteger();  
6. ...  
7. finish(() -> {  
8.     for (int i=0; i<numTasks; i++ )  
9.         async(() -> {  
10.            do {  
11.                j = a.getAndAdd(1);  
12.                // can also use a.getAndIncrement()  
13.                if (j >= X.length) break;  
14.                taskId[j] = i; // Task i processes string X[j]  
15.            ...  
16.        } while (true);  
17.    });  
18.}); // finish-for-async
```



# Work-Sharing Pattern using AtomicInteger

```
1. import java.util.concurrent.atomic.AtomicInteger;  
2. ...  
3. String[] X = ... ; int numTasks = ...;  
4. int[] taskId = new int[X.length];  
5. AtomicInteger a = new AtomicInteger();  
6. ...  
7. finish(() -> {  
8.     for (int i=0; i<numTasks; i++ )  
9.         async(() -> {  
10.            do {  
11.                int j = a.getAndAdd(1);  
12.                // can also use a.getAndIncrement()  
13.                if (j >= X.length) break;  
14.                taskId[j] = i; // Task i processes string X[j]  
15.            ...  
16.        } while (true);  
17.    });  
18.}); // finish-for-async
```



# 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 the 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



# Locking guarantees in Java

- It is preferable to use `java.util.concurrent.atomic` or `HJlib` isolated constructs, since they cannot deadlock
- Locks are needed for more general cases. Basic idea is for JVM to implement `synchronized(a) <stmt>` as follows:
  1. Acquire lock for object a
  2. Execute `<stmt>`
  3. Release lock for object a
- The responsibility for ensuring that the choice of locks correctly implements the semantics of isolation lies with the programmer.
- The main guarantee provided by locks is that only one thread can hold a given lock at a time, and the thread is blocked when acquiring a lock if the lock is unavailable.



# 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()");  
        ...  
        doSomething(); // Doesn't deadlock!  
    }  
}
```



# 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);  
            }  
        }  
    }  
}
```



# Deadlock avoidance in HJ with object-based isolation

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

```
public class ObviousDeadlock {  
    . . .  
    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);  
        }  
    }  
}
```

