SCHULICH School of Engineering

ENSF 607

3 – Multithreading (Part II)

Objects Lock

Intrinsic Locks

- Every object has an **intrinsic lock** associated with it.
- As long as a thread owns an **intrinsic lock**, **no other thread** can acquire the same lock.
- When a thread releases an intrinsic lock, **a happens-before relationship** is established.
  - A happens-before relationship is a relation between the result of two events, such that if one event should happen before another event, the result must reflect.

Intrinsic Locks Cont'd

- When a thread invokes a **synchronized method**, it automatically acquires the **intrinsic lock** for that method's object and releases it when the method returns.
  - **The lock release occurs even if the return was caused by an uncaught exception.**
- For **static synchronized**, the thread acquires the **intrinsic lock** for the Class object associated with the class.
  - **Access to class's static fields is controlled by a lock.**

Synchronized Statements

- To create a synchronized block (for example part of a method), synchronized statements can be used:

```
public void doSomething(int n) {
    synchronized(this) {
            n++;
    }

    doSomethingElse();
}
```

Synchronized Statements

- Consider the class **TwoLock** with two fields x, and y that you may assume are never used together:

```
public class TwoLock {
    private int x = 0, y = 0 ;
    private Object a = new Object();
    private Object b = new Object();

    public void inc1() {
        synchronized(a) {
            x++;
        }
    }
    public void inc2() {
        synchronized(b) {
            y++;
        }
    }
}
```

Reentrant Lock in Java

- Java class ReentrantLock provides a mechanism to lock an object while a thread has the ownership of the lock.
- `Import java.util.cuncurrent.locks.*;`
- Create an object of `ReentrantLock`:
  - **ReentrantLock mylock = new ReentrantLock();**
  - Then use the lock method to protect that portion of the code from multiple access of several threads:

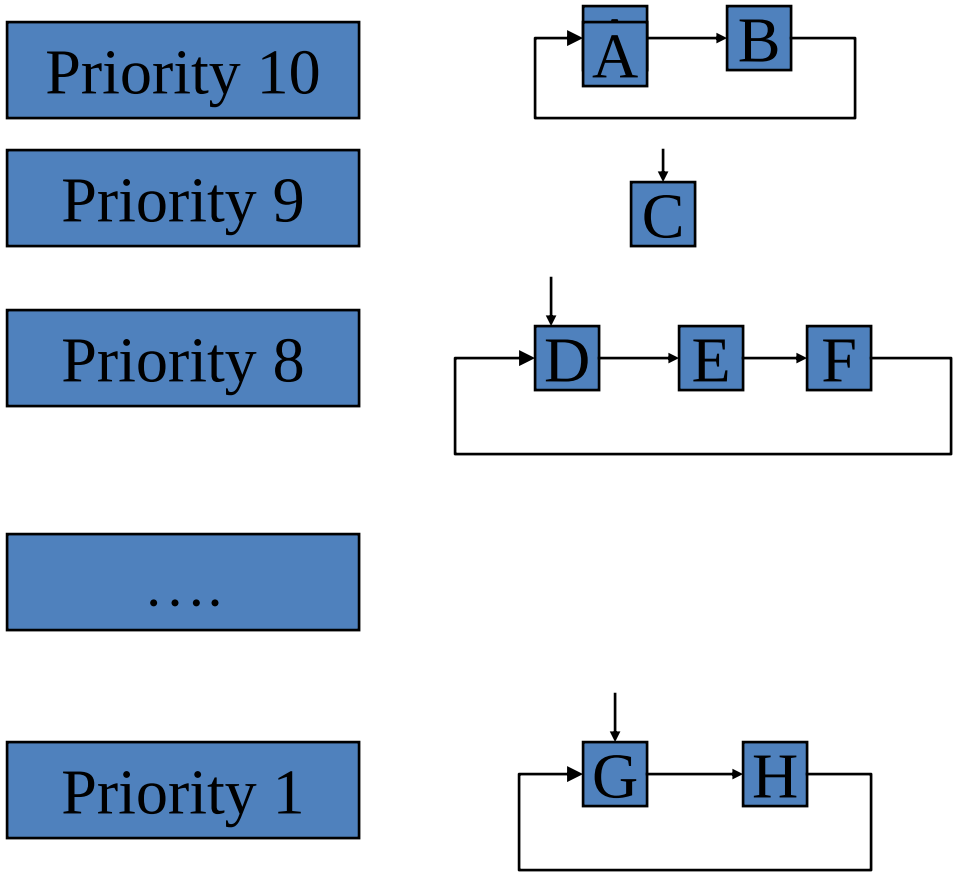
```
void fun() {
  myLock.lock();
  Try{
        // critical section
     }
  finally {
        myLock.unlock()
  }
}
```

Thread Scheduling

- The **priority** of a thread can be set by the function `setPriority ()`. This function accepts an integer value as its parameter (ranging from 1 to 10). 1 is the lowest and, 10 is the highest level of priority.

- The priority of a thread can be determined by calling `getPriority()`.

Thread Scheduling



From Deitel & Deitel (Third Ed.)

Java How to program, Prentice Hall

Object Methods: Wait and Notify

- ***wait*** and **notify** are defined in class object.
- Function ***wait*** can be used to force a thread to wait until the condition changes:

```
synchronized public void fun() {
    while (condition == false)
            wait();
     …
}
```

Wait and Notify

- Function **notify** can be used to wake up the thread (s) that are waiting for condition to change (**lock to be released**):

```
synchronized public void fun2() {
        …
        condition = true;
        notify();
    }
```

- Function **notifyAll** can be used to notify all the threads waiting for a condition to change.

Class Exercises

How Join and Some Other Methods Work

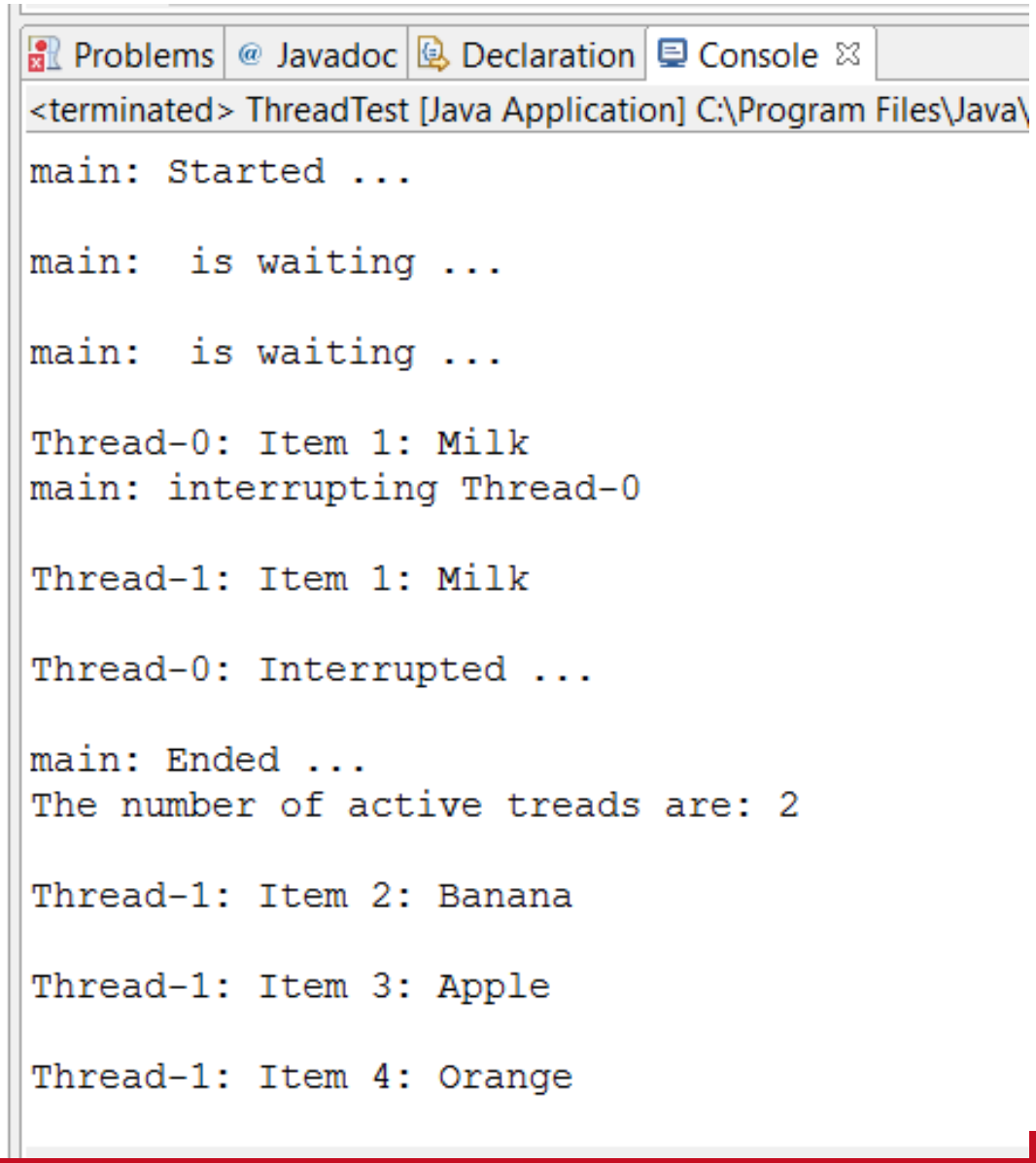
```
class ItemList implements Runnable {

    String items[] = { "Item 1: Milk", "Item 2: Banana",
                       "Item 3: Apple", "Item 4: Orange" };

    public void run() {
        try {
             for (int i = 0; i < items.length; i++) {
                 Thread.sleep(2000);
                 System.out.format("\n%s: %s%n",
                                   Thread.currentThread().getName()
                                   items[i]);
             }
        } catch (InterruptedException e) {
             System.out.format("\n%s: %s%n",
                               Thread.currentThread().getName(),
                               "Interrupted ...");
        }
    }
}
```

```
public class ThreadTest {
    public static void main(String args[]) throws InterruptedException {
        System.out.format("\n%s: %s%n", Thread.currentThread().getName(),
                          "Started ...");
        long startTime = System.currentTimeMillis();
        Thread t = new Thread(new ItemList());
        Thread t2 = new Thread(new ItemList());
        t.start();
        t2.start();
        while (t.isAlive()) {
            System.out.format("\n%s: %s%n", Thread.currentThread().getName(),
                              " is waiting ...");
            t.join(1000);
            if (((System.currentTimeMillis() - startTime) > 2000) &&
                  t.isAlive()) {
                System.out.format("%s: %s%n",
                                  Thread.currentThread().getName(),
                                  "interrupting " + t.getName());
                t.interrupt();
                break;
            } // END OF IF
        } // END OF WHILE
        Thread.sleep(100);
        System.out.format("\n%s: %s%n", Thread.currentThread().getName(),
                          "Ended ...");
        System.out.println("The number of active treads are: " +
                           Thread.activeCount());
    } // END OF MAIN
} // END OF CLASS
```

Program output:



A few terminologies:

- Deadlock:
  - Two or more thread reach to a situation that all get blocked forever.
- Starvation:
  - A thread cannot make progress because cannot gain access to shared resources because one or more "greedy" threads holding the resources for a long time
- Livelock:
  - If thread A needs response from thread B, and B needs response from thread C, then a livelock may happen if the threads cannot make any progress because of this situation.
- Completeness and Liveness:
  - A thread may reached to end of its code but it has to remain alive because of other threads.

Coordinating Threads

Exercise

- In a **client-server application**, a supplier supplies an item, and a shopper collects it when it is available.
- **Precondition for shopper:**
  - Wait until item arrives
- **Precondition for supplier:**
  - Wait until previous item is picked up

- **We will need to have two threads (supplier and shopper)**

```
import java.util.Random;
class Supplier implements Runnable {
      private Shipping shipping;

      public Supplier(Shipping shipping) {
         this.shipping = shipping;
      }

      public void run() {
         String itemList[] = { "IBM Laptop", "Samsung Galaxy S5", "Power Adapter",
                               "Music CD" };
         for (int i = 0; i < itemList.length; i++) {
              try {
                    shipping.set(itemList[i]);
                } catch (InterruptedException e1) {
                       e1.printStackTrace();
                }
                System.out.format("Item received from supplier: %s%n", itemList[i]);
                try {
                       Thread.sleep(new Random().nextInt(7000));
                } catch (InterruptedException e) {}
         }
         try {
               shipping.set("NONE");
         } catch (InterruptedException e) {
                e.printStackTrace();
         }
     }
}
```

```
import java.util.Random;

class Shopper implements Runnable {
        private Shipping shipping;

        public Shopper(Shipping shipping) {
           this.shipping = shipping;
        }

        public void run() {
            while (true) {
                String item = null;
                try {
                        item = shipping.get();
                } catch (InterruptedException e1) {
                        e1.printStackTrace();
                }
                if (item.equals("NONE"))
                break;
                System.out.format("Item picked up by shopper: %s%n", item);
                try {
                        Thread.sleep(new Random().nextInt(7000));
                } catch (InterruptedException e) {}
           }
        }
}
```

```
class Shipping {
    private String item; // item sent from supplier to shopper.
    private boolean empty = true; // true if no item in shipping

    public synchronized String get() throws InterruptedException {
            while (empty) { // shopper waits while empty
                  wait();
            }
            empty = true;
            notifyAll(); // Notify suppliers -- ready to get
            return item;
    }

    public synchronized void set(String item) throws InterruptedException {

            while (!empty) { // supplier waits if it's full
                  wait();
            }
            empty = false;
            this.item = item;
            notifyAll(); // Notify shopper that item is available
    }
}
```

```
public class Demo {
    public static void main(String[] args) throws InterruptedException
    {
        Shipping shipping = new Shipping();
        Supplier sup = new Supplier(shipping);
        Thread t1 = new Thread(sup);
        Shopper sh = new Shopper(shipping);
        Thread t2 = new Thread(sh);
        t1.start();
        t2.start();
    }
}
```

Program output

Item received from supplier: IBM Laptop

Item picked up by shopper: IBM Laptop

Item received from supplier: Samsung Galaxy S5

Item picked up by shopper: Samsung Galaxy S5

Item received from supplier: Power Adapter

Item picked up by shopper: Power Adapter

Item received from supplier: Music CD

Item picked up by shopper: Music CD