Monitors

CSCI 201
Principles of Software Development

Jeffrey Miller, Ph.D.
jeffrey.miller@usc.edu
Outline

• Monitors
• Program
Monitor Overview

- A **monitor** is an object with mutual exclusion and synchronization capabilities
  - All objects in Java can be monitors (see **Object API** on next slide)
- The **synchronized** keyword enables the use of monitors
  - Methods or individual blocks of code in Java can be synchronized
- A thread enters the monitor by acquiring a **lock** on it and exits by releasing the lock
- An object becomes a monitor once a thread locks it using the **synchronized** keyword
- A thread can call **wait()** inside a monitor, which will release the lock on the object
  - That thread must then be awakened using **notify()** or **notifyAll()** from another thread to be moved back into the Ready state
## Object class

### Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected <code>Object</code></td>
<td><code>clone()</code> Creates and returns a copy of this object.</td>
</tr>
<tr>
<td>boolean</td>
<td><code>equals(Object obj)</code> Indicates whether some other object is &quot;equal to&quot; this one.</td>
</tr>
<tr>
<td>protected <code>void</code></td>
<td><code>finalize()</code> Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.</td>
</tr>
<tr>
<td><code>Class&lt;?&gt;</code></td>
<td><code>getClass()</code> Returns the runtime class of this <code>Object</code>.</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>hashCode()</code> Returns a hash code value for the object.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><code>notify()</code> Wakes up a single thread that is waiting on this object's monitor.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><code>notifyAll()</code> Wakes up all threads that are waiting on this object's monitor.</td>
</tr>
<tr>
<td><code>String</code></td>
<td><code>toString()</code> Returns a string representation of the object.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><code>wait()</code> Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><code>wait(long timeout)</code> Causes the current thread to wait until either another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or a specified amount of time has elapsed.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><code>wait(long timeout, int nanos)</code> Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.</td>
</tr>
</tbody>
</table>
**synchronized Keyword**

- The `synchronized` keyword puts a restriction on a method or block of code that only one thread can be inside that method or block at a time
  - No other thread will be able to enter that method or block of code if another thread is currently executing inside of it, regardless of whether it is in the CPU currently or not

- Before a block of `synchronized` code can execute, a lock must be obtained
  - A lock is a binary mechanism for exclusive use of a resource
  - Locks can only be acquired by one object at a time
synchronized Methods

- **synchronized Non-Static Methods**
  - The lock obtained is on the *object* on which the method was invoked.
  - When a thread invokes a *synchronized* instance method of an object, the lock of that object is acquired first, then the method is executed, then the lock is released.
    - Another thread invoking any *synchronized* method or block of code *on that object* is blocked until the lock is released.

- **synchronized static Methods**
  - The lock obtained is on the *class* on which the method was invoked (even if the method was invoked from an instance of the class, which would be bad programming).
  - When a thread invokes a *synchronized static* method of a class, the lock on that class is acquired first, then the method is executed, then the lock is released.
    - Another thread invoking any *synchronized* method or block of code *on that class* is blocked until the lock is released.
Synchronization Example #1

1   class SyncClass {
2     synchronized void foo() {
3       // foo line 1
4       // foo line 2
5     }
6     synchronized void bar() {
7       // bar line 1
8       // bar line 2
9     }
10    void meth() {
11      // meth line 1
12      // meth line 2
13    }
14  }
15
16  public class MainClass {
17    public static void main(String [] args) {
18      SyncClass sc = new SyncClass();
19      // multiple threads created
20    }
21  }

Thread T1 calls \texttt{sc.foo();} and gets switched out of the CPU after line 3

Thread T2 calls \texttt{sc.foo();}

Will T2 be able to execute?

Not until T1 releases the lock on \texttt{sc}
Synchronization Example #2

```java
class SyncClass {
    synchronized void foo() {
        // foo line 1
        // foo line 2
    }
    synchronized void bar() {
        // bar line 1
        // bar line 2
    }
    void meth() {
        // meth line 1
        // meth line 2
    }
}

public class MainClass {
    public static void main(String[] args) {
        SyncClass sc = new SyncClass();
        // multiple threads created
    }
}
```

Thread T1 calls `sc.foo();` and gets switched out of the CPU after line 3.

Thread T2 calls `sc.bar();`

Will T2 be able to execute?

Not until T1 releases the lock on `sc`
Synchronization Example #3

```java
1  class SyncClass {
2     synchronized void foo() {
3       // foo line 1
4       // foo line 2
5     }
6     synchronized void bar() {
7       // bar line 1
8       // bar line 2
9     }
10    void meth() {
11      // meth line 1
12      // meth line 2
13    }
14  }
15
16  public class MainClass {
17    public static void main(String [] args) {
18      SyncClass sc = new SyncClass();
19      SyncClass sc2 = new SyncClass();
20      // multiple threads created
21    }
22  }
```

Thread T1 calls `sc.foo();` and gets switched out of the CPU after line 3

Thread T2 calls `sc2.foo();`

Will T2 be able to execute?

Yes, since T1 acquires the lock on `sc` and T2 acquires the lock on `sc2`
Synchronization Example #4

```java
1 class SyncClass {
2   static synchronized void foo() {
3     // foo line 1
4     // foo line 2
5   }
6   static synchronized void bar() {
7     // bar line 1
8     // bar line 2
9   }
10  void meth() {
11     // meth line 1
12     // meth line 2
13  }
14 }
15
16 public class MainClass {
17   public static void main(String [] args) {
18     SyncClass sc = new SyncClass();
19     SyncClass sc2 = new SyncClass();
20     // multiple threads created
21   }
22 }
```

Thread T1 calls `sc.foo();` and gets switched out of the CPU after line 3

Thread T2 calls `sc2.bar();`

Will T2 be able to execute?

Not until T1 releases the lock on SyncClass
Synchronization Example #5

```java
1   class SyncClass {
2     static synchronized void foo() {
3       // foo line 1
4       // foo line 2
5     }
6     static synchronized void bar() {
7       // bar line 1
8       // bar line 2
9     }
10    void meth() {
11      // meth line 1
12      // meth line 2
13    }
14  }
15
16  public class MainClass {
17    public static void main(String [] args) {
18      SyncClass sc = new SyncClass();
19      SyncClass sc2 = new SyncClass();
20      // multiple threads created
21    }
22  }
```

Thread T1 calls `SyncClass.foo();` and gets switched out of the CPU after line 3.

Thread T2 calls `SyncClass.bar();`

Will T2 be able to execute?

Not until T1 releases the lock on `SyncClass`
Synchronization Example #6

```java
1  class SyncClass {
2      static synchronized void foo() {
3         // foo line 1
4         // foo line 2
5      }
6      synchronized void bar() {
7         // bar line 1
8         // bar line 2
9      }
10     void meth() {
11        // meth line 1
12        // meth line 2
13     }
14  }
15
16  public class MainClass {
17      public static void main(String [] args) {
18         SyncClass sc = new SyncClass();
19         // multiple threads created
20      }
21  }
```

Thread T1 calls `SyncClass.foo();` and gets switched out of the CPU after line 3

Thread T2 calls `sc.bar();`

Will T2 be able to execute?

Yes, since T1 has the lock on `SyncClass` and T2 has the lock on `sc`
Synchronization Example #7

```java
1   class SyncClass {
2     static synchronized void foo() {
3       // foo line 1
4       // foo line 2
5     }
6     synchronized void bar() {
7           meth();
8       // bar line 2
9     }
10    void meth() {
11      // meth line 1
12      // meth line 2
13    }
14  }
15
16  public class MainClass {
17    public static void main(String [] args) {
18      SyncClass sc = new SyncClass();
19      // multiple threads created
20    }
21  }
```

Thread T1 calls `sc.bar();` and gets switched out of the CPU after line 11 in `meth()`

Thread T2 calls `sc.meth();`

Will T2 be able to execute?

Yes, since T1 has the lock on `sc`
And T2 doesn’t need a lock
synchronized Statements

- We do not need to synchronize entire methods if only a part of the method needs to be synchronized.
- A `synchronized` statement can be used to acquire a lock on any object, not just the current object, or on a class.
  ```java
  synchronized(obj) {
      // synchronized code
  }
  synchronized(String.class) {
      // synchronized code
  }
  ```
- The lock would have to be obtained on the object `obj` or the class before the code in that block could execute.
  - If the lock cannot be obtained, the thread will block at that line until it can obtain the lock.
- Note that any `synchronized` method can be converted into a `synchronized` block of code.
  ```java
  public synchronized void meth() {
      // code
  }
  ```
  ```java
  public void meth() {
      synchronized(this) {
          // code
      }
  }
  ```
AddAPenny Example Revisited

1   import java.util.concurrent.ExecutorService;
2   import java.util.concurrent.Executors;
3
4   public class AddAPenny implements Runnable {
5     private static PiggyBank piggy = new PiggyBank();
6
7     public void run() {
8       piggy.deposit(1);
9     }
10
11    public static void main(String [] args) {
12      ExecutorService executor = Executors.newCachedThreadPool();
13      for (int i=0; i < 100; i++) {
14        executor.execute(new AddAPenny());
15      }
16      executor.shutdown();
17      // wait until all tasks are finished
18      while(!executor.isTerminated()) {
19        Thread.yield();
20      }
21      System.out.println("Balance = " + piggy.getBalance());
22    }
23
24  }
25
26  class PiggyBank {
27    private int balance = 0;
28    public int getBalance() {
29      return balance;
30    }
31    public void deposit(int amount) {
32      int newBalance = balance + amount;
33      Thread.yield();
34      balance = newBalance;
35    }
36  }

4 Executions
AddAPenny with Synchronization

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class AddAPenny implements Runnable {
  private static PiggyBank piggy = new PiggyBank();

  public void run() {
    piggy.deposit(1);
  }

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();
    for (int i=0; i < 100; i++) {
      executor.execute(new AddAPenny());
    }
    executor.shutdown();
    // wait until all tasks are finished
    while(!executor.isTerminated()) {
      Thread.yield();
    }
    System.out.println("Balance = " + piggy.getBalance());
  }
}

class PiggyBank {
  private int balance = 0;
  public int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    int newBalance = balance + amount;
    Thread.yield();
    balance = newBalance;
  }
}
```

Outline

• Monitors
• Program
Program

- Download the AddAndRemoveAPenny code from the course web site and execute it
  - Make sure you understand why the output is what it is
- What modification could you make to the code to force it to hang if the total amount of withdrawals exceeds the total amount of deposits?
- Modify the code to remove having an equal number of threads that withdraw and deposit
  - Does the code always terminate in either case?
  - How can you make the code always terminate?