Monitors

CSCI 201
Principles of Software Development

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

USC Viterbi
School of Engineering

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 has the monitor functionality invoked once a thread locks it using the `synchronized` keyword

`Object` class

Method Summary

Methods

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

Monitor Overview

- The `Object` class has methods on the monitor that can be called, though this should be used cautiously
  - Monitor functionality is implemented in the `synchronized` keyword, and calling monitor methods directly may produce different results when synchronizing the object
- 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

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

Thread T2 calls `sc.foo();`

Will T2 be able to execute?

Not until T1 releases the lock on `sc`

Synchronization Example #2

```
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 `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

```
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

```
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

```
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

```
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

```
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

```
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

```
public synchronized void meth() {
    // code
}
```

```
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
22      System.out.println("Balance = " + piggy.getBalance());
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

Console | Problems

```
<terminated> Test [Java Application]
Balance = 4
```

Console | Problems

```
<terminated> Test [Java Application]
Balance = 6
```

Console | Problems

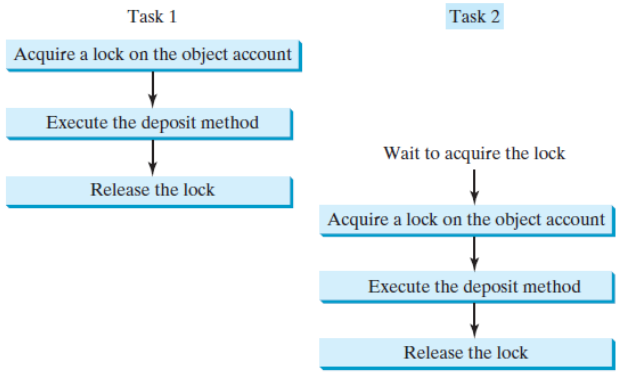
```
<terminated> Test [Java Application]
Balance = 7
```

Console | Problems

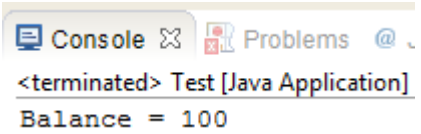
```
<terminated> Test [Java Application]
Balance = 10
```

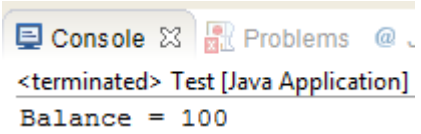
AddAPenny with Synchronization

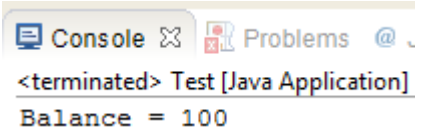
```
1   import java.util.concurrent.ExecutorService;
2   import java.util.concurrent.Executors;
3
4   public class AddAPenny implements Runnable {
5     private static PiggyBank piggy = new PiggyBank();
6
7     public void run() {
8       piggy.deposit(1);
9     }
10
11    public static void main(String [] 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
22      System.out.println("Balance = " + piggy.getBalance());
23    }
24  }
25
26  class PiggyBank {
27    private int balance = 0;
28    public int getBalance() {
29      return balance;
30    }
31    public synchronized void deposit(int amount) {
32      int newBalance = balance + amount;
33      Thread.yield();
34      balance = newBalance;
35    }
36  }
```

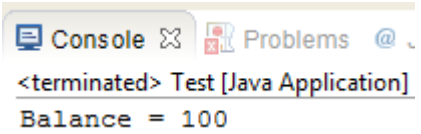


4 Executions









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?