Monitors

CSCI 201L

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

HTTP://WWW-SCF.USC.EDU/~CSCI201
Outline

- Monitors
Monitor Overview

- A monitor is an object with mutual exclusion and synchronization capabilities
  - All objects in Java can be monitors
- 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
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

```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         // 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

```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      // 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

```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();
    SyncClass sc2 = new SyncClass();
    // multiple threads created
  }
}
```

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`
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
class SyncClass {
    static synchronized void foo() {
        // foo line 1
        // foo line 2
    }
    static 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();
        SyncClass sc2 = new SyncClass();
        // multiple threads created
    }
}
```

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
class SyncClass {
    static 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();
        SyncClass sc2 = new SyncClass();
        // multiple threads created
    }
}
```

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

Thread T2 calls sc2.bar();

Will T2 be able to execute?

Yes, since T1 has the lock on SyncClass
And T2 has the lock on sc2
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
  
  ```java
  synchronized(obj) {
      // synchronized code
  }
  ```

- The lock would have to be obtained on the object `obj` 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

```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 void deposit(int amount) {
        int newBalance = balance + amount;
        Thread.yield();
        balance = newBalance;
    }
}
```

4 Executions

- Balance = 4
- Balance = 6
- Balance = 7
- Balance = 10
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 website 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?