Concurrency
CSCI 201L

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

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

- Synchronization
  - Critical Section
Motivation for Synchronization

- Thread synchronization is used to coordinate the execution of dependent threads
  - Sometimes we want to have some control over when threads execute or what code they are able to simultaneously access

- Shared resources pose potential problems to multi-threaded code since we don’t typically have control over when a thread will be switched into and out of the CPU
Synchronization Example #1

1. public class AddAPenny implements Runnable {
2.     private static PiggyBank piggy = new PiggyBank();
3. 
4.     public void run() {
5.         piggy.deposit(1);
6.     }
7. 
8.     public static void main(String[] args) {
9.         for (int i=0; i < 100; i++) {
10.            Thread t = new Thread(new AddAPenny());
11.            t.start();
12.        }
13.        System.out.println("Balance = " + piggy.getBalance());
14.     }
15. }
16. 
17. class PiggyBank {
18.     private int balance = 0;
19.     public int getBalance() {
20.         return balance;
21.     }
22.     public void deposit(int amount) {
23.         int newBalance = balance + amount;
24.         Thread.yield();
25.         balance = newBalance;
26.     }
27. }

4 Executions

![Console](problems.png)

<terminated> Test [Java Application]
Balance = 9

<terminated> Test [Java Application]
Balance = 6

<terminated> Test [Java Application]
Balance = 12
Synchronization Example #2

```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

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

<terminated> Test [Java Application]  
Balance = 6

<terminated> Test [Java Application]  
Balance = 7

<terminated> Test [Java Application]  
Balance = 10
```
Synchronization Example #3

```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) {
        balance += amount;
    }
}
```

4 Executions

![console output](image)
The problem in the previous examples is based on the use of a shared variable across multiple threads (the PiggyBank object `piggy`)

- The OS switches threads out of the CPU at times unknown to us
- This seems to be happening inside the `deposit(int)` method after `newBalance` is set but before it is assigned back to `balance`
- This causes the old value of `balance` to be used in subsequent threads

When we reduce the `deposit(int)` method to a single statement, we get higher values for `balance` because there are fewer locations at which the OS can switch the thread out of the CPU before setting the value of `balance`
Race Conditions

- A **race condition** is a state where more than one thread is accessing the same variable and overwriting the other thread’s updates.
  - This is a common problem in multi-threaded programming.

- A class is **thread-safe** if an object of the class does not cause a race condition in the presence of multiple threads.
  - The AddAPenny class in the previous examples is not thread-safe.
Critical Section

- To avoid race conditions, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program
  - This part of the program is called a critical section or critical region
- In the previous examples, the critical section would be the `deposit(int)` method in the PiggyBank class
  - If only one thread at a time was able to access the `deposit(int)` method, we would not have a race condition on the shared variable balance
    - Note that balance is shared in the static instance of PiggyBank

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