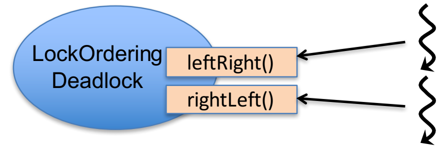
Lock-ordering Deadlocks

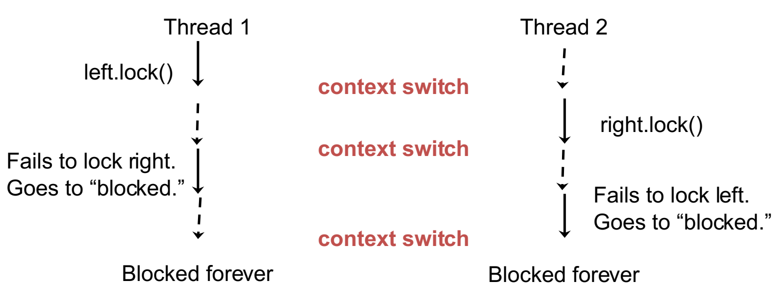
Lock-ordering Deadlocks



Threads can deadlock.

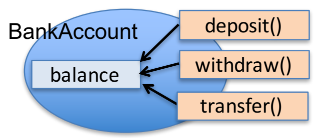
```
• Class LockOrderingDeadlock{
   private ReentrantLock left = new ReentrantLock();
   private ReentrantLock right = new ReentrantLock();
   public void leftRight(){
       left.lock();
       right.lock();
       // atomic code
       right.unlock();
       left.unlock();
   }
   public void rightLeft(){
       right.lock();
       left.lock();
       // atomic code
       left.unlock();
       right.unlock();
   }
```

```
• Class LockOrderingDeadlock{
   private ReentrantLock left = new ReentrantLock();
   private ReentrantLock right = new ReentrantLock();
   public void leftRight(){
       left.lock();
       right.lock();          A context switch can occur here.
       // atomic code
       right.unlock();
       left.unlock(); }
   public void rightLeft(){
       right.lock();
       left.lock();           A context switch can occur here.
       // atomic code
       left.unlock();
       right.unlock(); }
```



- Problem:
  - Threads try to acquire *the same set of locks* in *different orders*.
    - *Inconsistent* lock ordering
      - Thread 1: left → right
      - Thread 2: right → left
- To-do:
  - Have all threads acquire the locks in a globally-fixed order.
- **Be careful when you use multiple locks in order!**

Dynamic Lock-ordering Deadlocks

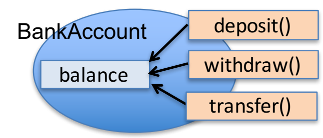



```
class BankAccount{
    private ReentrantLock lock;
    ...
    public void deposit(...)
    public void withdraw(...)
    public void transfer(...)
}
```

```
void deposit(double amount){
     lock.lock();
     balance += amount;
     lock.unlock(); }
```

```
void withdraw(double amount){
     lock.lock();
     balance -= amount;
     lock.unlock(); }
```

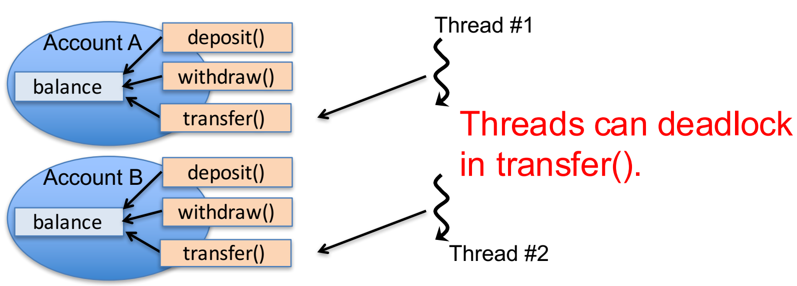
```
void void transfer(Account destination, double amount){
     lock.lock();
     if( balance < amount )
         // generate an error msg or throw an exception
     else{
         withdraw(amount);              // Nested locking. No problem.
         destination.deposit(amount);  // Acquire another lock.
     }
     lock.unlock();}
```




```
class BankAccount{
    private ReentrantLock lock;
    ...
    public void deposit(...)
    public void withdraw(...)
    public void transfer(...)
}
```

```
void transfer(Account destination, double amount){
     lock.lock();
     if( balance < amount )
         // generate an error msg or throw an exception
     else{
         withdraw(amount);              // Nested locking. No problem.
         destination.deposit(amount);  // Acquire another lock.
     }
     lock.unlock();}
```

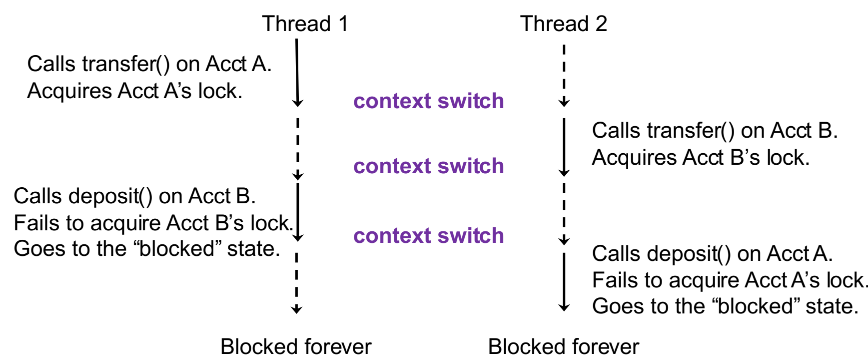
- It looks as if all threads acquire the two locks (this.lock and destination.lock) in the same order.
- However, this code can have a lock-ordering deadlock.



- Imagine a scenario where
  - a thread (#1) transfers money from Account A to B
  - another thread (#2) transfers money from B to A.

```
public void transfer(Account destination, double amount){
     lock.lock();
     if( balance < amount )
         // generate an error msg or...
     else{
         withdraw(amount);
         destination.deposit(amount);
     }
     lock.unlock();}
```

A context switch can occur here.



- Problem
  - Threads try to acquire *the same set of locks* in *different orders*.
    - *Inconsistent* lock ordering.
      - Thread 1: Acct A's lock → Acct B's lock
      - Thread 2: Acct B's lock → Acct A's lock
  - This can occur with bad timing although code looks OK.
    - A -> B and C -> D at the same time (No lock-ordering deadlock)
    - A -> B and B -> A at the same time (Possible lock-ordering deadlock)
- **Be careful when you use multiple locks in order!!!**

Solutions

- Static lock
- Timed locking
- Ordered locking
- Nested tryLock()

Solution 1: Static Lock

```
• private static ReentrantLock lock = new ReentrantLock();

• public void deposit(double amount){
      lock.lock();
      this.balance += amount;
      lock.unlock(); }

• public void transfer(Account destination, double amount){
      lock.lock();
      if( this.balance < amount )
         // generate an error msg or throw an exception
      else{
         this.withdraw(amount);        // Nested locking
         destination.deposit(amount);  // Nested locking!
      }
      lock.unlock();}
```

- Pros
  - Simple solution
- Cons
  - Performance penalty
    - Transfers on different accounts are performed sequentially (not concurrently).
    - Deposit operations on different accounts are performed sequentially (not concurrently).
    - Withdrawal operations on different accounts are performed sequentially (not concurrently).

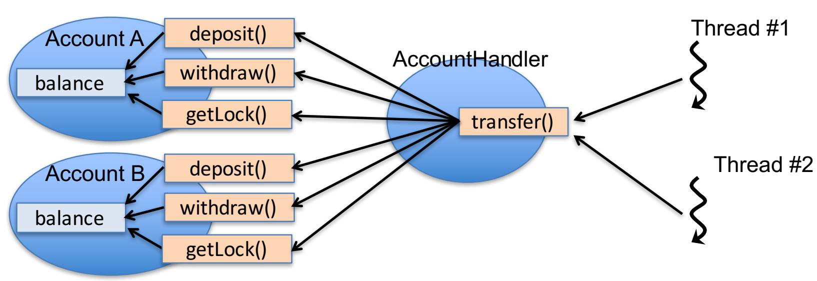
Solution 2: Timed Locking

```
• public void deposit(double amount){
      if( !lock.tryLock(3, TimeUnit.SECONDS) ){
          // generate an error msg or throw an exception
      }else{
          this.balance += amount;
          lock.unlock();} }

• public void transfer(Account destination, double amount){
      lock.lock();
      if( this.balance < amount )
          // generate an error msg or throw an exception
      else{
          this.withdraw(amount);
          destination.deposit(amount);
      }
      lock.unlock();} //Make sure to release this lock when
                      // an error/exception occurs.
```

- Pros
  - Simple solution
  - More efficient than Solution #1
    - By using a non-static lock
- Cons
  - Transfers and deposits might never be completed.
    - May look like unprofessional.

Solution 3: Ordered Locking




```
• public void transfer( Account source,
                        Account destination,
                        double amount){
      if( source.getAcctNum() < destination.getAcctNum() ){
          source.getLock().lock();
          destination.getLock().lock();
          if( source.getBalance() < amount )
              // generate an error msg or throw an exception
          else{
              source.withdraw(amount);     //Nested locking
              destination.deposit(amount); //Nested locking
          }
          destination.getLock().unlock();
          source.getLock().unlock();
      }
      else if( source.getAcctNum() > destination.getAcctNum() ){
          destination.getLock().lock();
          source.getLock().lock();
          ...
          source.getLock().unlock();
          destination.getLock().unlock();
      }
```

- Pros
  - Locks are always acquired in the same order.
  - More efficient than Solution #1
    - By using a non-static lock
  - More professional than Solution #2
    - Transfers and deposits complete for sure.
- Cons
  - Using an application-specific/dependent data.
  - Account numbers should not be changed after accounts are set up.
    - If you allow dynamic changes of account numbers, you need to use an extra lock.

Solution 3a: Ordered Locking with Instance IDs

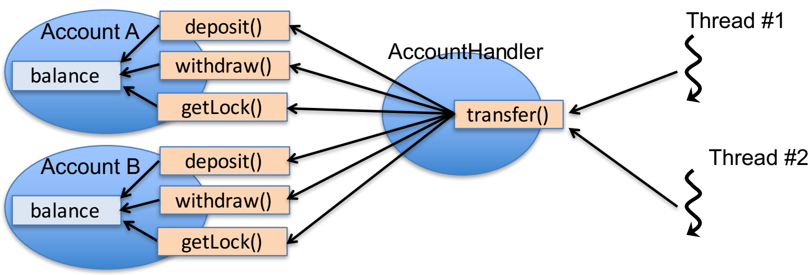
- Instance IDs
  - Unique IDs (hash code) that the local JVM assigns to individual class instances.
    - Unique and intact on the same JVM
      - 2 instances of the same class have different IDs.
      - No instances share the same ID.
      - IDs never change after they are assigned to instances.
  - Use System.identifyHashCode()

```
• public void transfer( Account source,
                        Account destination,
                        double amount){
      int sourceID = System.identifyHashCode(source);
      int destID = System.identifyHashCode(destination);

      if( sourceID < destID ){
         source.getLock().lock();
         destination.getLock().lock();
         if( source.getBalance() < amount )
            // generate an error msg or throw an exception
         else{
            source.withdraw(amount);     //Nested locking
            destination.deposit(amount); //Nested locking
         }
         destination.getLock().unlock();
         source.getLock().unlock();
      }
      if( sourceID > destID ){
         destination.getLock().lock();
         source.getLock().lock();
         ...
         source.getLock().unlock();
         destination.getLock().unlock();
      }
```

- Pros
  - Locks are always acquired in the same order.
  - More efficient than Solution #1
    - By using a non-static lock
  - More professional than Solution #2
    - Transfers and deposits complete for sure.
  - No application-specific data (e.g., account numbers) are necessary to order locking.
- Cons
  - N/A

Solution 4: Nested tryLock()



- Use nested tryLock() calls to implement an ALL-OR-NOTHING policy.
  - Acquire both of A's and B's locks, OR
  - Acquire none of them.
- Avoid a situation where a thread acquires one of the two locks and fails to acquire the other.

```
public void transfer(Account source,
                     Account destination,
                     double amount){
 Random random = new Random();

 while(true){
     if( source.getLock().tryLock() ){
         try{
             if( destination.getLock().tryLock() ){
                 try{
                     if( source.getBalance() < amount )
                         // generate an error msg/exception
                     else{
                         source.withdraw(amount);
                         destination.deposit(amount);
                     }
                     return;
                 }finally{
                     destination.getLock().unlock();
                 }
             }
         }finally{
             source.getLock().unlock();
         }
     }
     Thread.sleep(random.nextInt(1000));
 }
}
```

- If the first tryLock() fails, then sleep.
- If the first tryLock() succeeds but the second one fails, unlock the first lock and sleep.

- Pros
  - Locks are always acquired in the same order.
  - More efficient than Solution #1
    - By using a non-static lock
  - More professional than Solution #2
    - Transfers and deposits complete for sure.
  - No application-specific data (e.g., account numbers) are necessary to order locking.
- Cons
  - N/A in principle
  - Maybe not that simple?

HW 21 [Optional]

- Complete thread-safe code with Solutions 1, 2, 3, 3a and 4.
  - Modify ThreadSafeBankAccount2.java