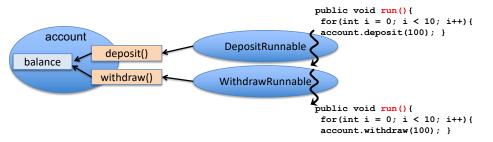
Thread Safety Issues

- Race conditions
- Deadlocks
- Thread-safe code is free from both race conditions and deadlocks.

Deadlock

DeadlockedBankAccount.java

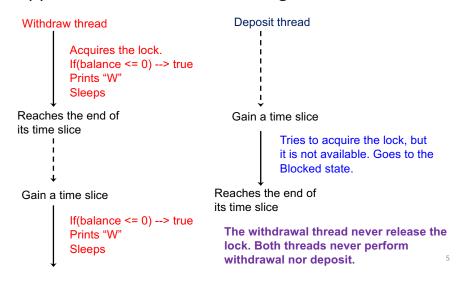


DeadlockedBankAccount.java

```
public void run(){
                                                     for (int i = 0; i < 10; i++) {
                                                     account.deposit(100); }
   account
                                    DepositRunnable
              deposit()
balance
             withdraw()
                                   WithdrawRunnable
 withdraw(double amount) {
                                                    public void run() {
  lock.lock();
                                                     for (int i = 0; i < 10; i++) {
  while (balance <= 0) {
                                                     account.withdraw(100); }
      System.out.print("W");
      // waiting for the balance to exceed 0
      Thread.sleep(1000);
  balance -= amount;
  lock.unlock(); }
 deposit(double amount) {
  lock.lock();
  while(balance >= 300) {
      System.out.print("W");
      // waiting for the balance to go below 300
      Thread.sleep(1000)
  balance += amount;
  lock.unlock(); }
```

How Can a Deadlock Occur?

Suppose the withdrawal thread goes ahead.



Note

- A JVM can perform context switches even when a thread runs atomic code.
 - A lock guarantees that only one thread exclusively runs atomic code at a time.
 - It does NOT control when to (or when not to) perform context switches.
 - Some resources explicitly/implicitly say that context switches never occur when a thread runs atomic code.
 - It is WRONG!

DeadlockedBankAccount2.java

- Previous version
 - withdraw(double amount) {
 lock.lock();
 while(balance <= 0) {
 System.out.print("W");
 Thread.sleep(1000);
 }
 balance -= amount;
 lock.unlock();
 }
 deposit(double amount) {
 lock.lock();
 while(balance >= 300) {
 System.out.print("W");
 Thread.sleep(1000)
 }
 balance += amount;
 lock.unlock();
- New version

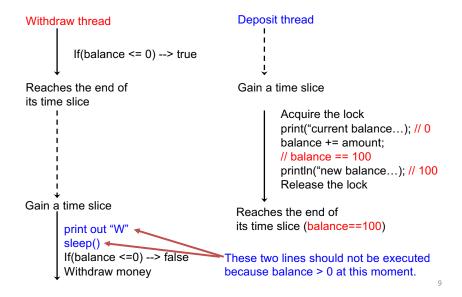
```
- withdraw(double amount) {
    while( balance <= 0 ) {
        System.out.print("W");
        Thread.sleep(1000);
    }
    lock.lock();
    balance -= amount;
    lock.unlock();
}

- deposit(double amount) {
    while( balance >= 300 ) {
        System.out.print("W");
        Thread.sleep(1000);
    }
    lock.lock();
    balance += amount;
    lock.unlock();
}
```

- Has no deadlock problems.
- Can generate race conditions.

7

A Potential Race Condition in DeadlockedBankAccount2



Avoiding Deadlocks and Race Conditions

- Use a condition Object(s).
 - Allows a thread to
 - Temporarily release a lock so that another thread can acquire it and proceed.
 - Re-acquire the lock later.
- java.util.concurrent.locks.Condition
 - Use a lock to obtain a condition instance

10

12

States of a Thread

```
notify()
                                                                       Timed
                 Waiting
                                  notifyAll()
                                                        sleep()
                                                                      Waiting
                                  signal()
                                                        join()
                                                                                notify()
                                  signalAll()
                                                        wait()
                                                                                notifyAll()
                                   interrupt()
                                                        await()
                                                                                signal()
                     sleep(
                                                                                signalAll()
                     join()
                                                                                interrupt()
                     wait()
                     await()
                                                                  Exits run() or
new Thread(...)
                                                                  Explicit thread
                          start()
                                                                  termination
              New
                                          Runnable
                                                                                 Terminated
                I/O op completion.
                                                        I/O operation or
                                                        wait for thread sync (lock
                Thread interruption.
                                                        acquisition)
                or thread sync (lock
                                           Blocked
                acquisition) completed
```

11

ThreadSafeBankAccount2.java

```
    Condition sufficientFundsCondition = lock.newCondition();

  Condition belowUpperLimitFundsCondition = lock.newCondition();

    withdraw(double amount) {

   lock.lock();
   while (balance <= 0) {
       // Wait for the balance to exceed 0
      sufficientFundsCondition.await(); }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }
  deposit(double amount) {
   lock.lock();
   while (balance >= 300) {
      // Wait for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
```

ThreadSafeBankAccount2.java

```
Condition sufficientFundsCondition = lock.newCondition();
Condition belowUpperLimitFundsCondition = lock.newCondition();
withdraw(double amount) {
 lock.lock();
 while (balance <= 0) {
      // Wait for the balance to exceed 0
     sufficientFundsCondition.await();
                                                  A "deposit" thread calls
 balance -= amount:
                                                  signalAll() to wake up
 belowUpperLimitFundsCondition.signalAll();
                                                  a thread(s) that is/are
 lock.unlock(); }
                                                  waiting until
                                                  balance > 0.
deposit(double amount) {
 lock.lock();
 while(balance >= 300) {
     // Wait for the balance to go below 300.
    belowUpperLimitFundsCondition.await(); }
 balance += amount;
 sufficientFundsCondition.signalAll() +
 lock.unlock(); }
```

ThreadSafeBankAccount2.java

```
Condition sufficientFundsCondition = lock.newCondition();
Condition belowUpperLimitFundsCondition = lock.newCondition();
withdraw(double amount) {
 lock.lock();
 while (balance <= 0) {
     // Wait for the balance to exceed 0
    sufficientFundsCondition.await();
 balance -= amount;
 belowUpperLimitFundsCondition.signalAll() =
 lock.unlock(); }
deposit(double amount) {
 lock.lock();
 while(balance >= 300) {
    // Wait for the balance to go below 300.
    belowUpperLimitFundsCondition.await();
 balance += amount;
 sufficientFundsCondition.signalAll() +
 lock.unlock(); }
```

ThreadSafeBankAccount2.java

Condition sufficientFundsCondition = lock.newCondition();

```
Condition belowUpperLimitFundsCondition = lock.newCondition();

    withdraw(double amount) {

   lock.lock();
   while(balance <= 0){
        // Wait for the balance to exceed 0
       sufficientFundsCondition.await();
}
   balance -= amount:
   belowUpperLimitFundsCondition.signalAll();
                                                    A "withdraw" thread
   lock.unlock(); }
                                                    calls signalAll() to
                                                    wake up a thread(s)
  deposit(double amount) {
                                                    that is/are waiting until
   lock.lock();
                                                    balance < 300.
   while(balance >= 300) {
       // Wait for the balance to go below 300.
      belowUpperLimitFundsCondition.await();
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
```

Condition

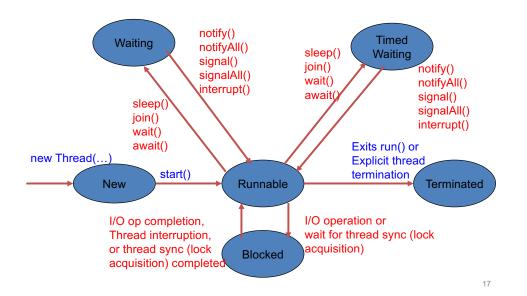
await()

- Will wait until it is signaled or interrupted
- Will wait until it is signaled or interrupted, or until a specified waiting time (relative time) elapsed.
- Will wait until it is signaled or interrupted, or until a specified deadline (absolute time).
- goes to the Runnable state and re-acquires a lock, if signaled.
 - Will be "blocked" if the thread re-acquisition fails.
- Throws an InterruptedException, if interrupted.
 - c.f. A previous lecture note on thread interruption

signalAll()

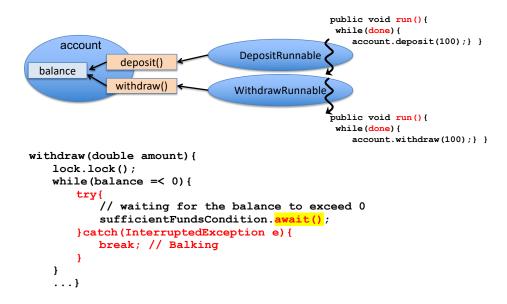
- Wakes up all waiting threads on a condition object.
 - All of them go to the "runnable" state.
 - One of them will re-acquire a lock.

States of a Thread



- When a thread calls await(), signalAll() or signal() on a Condition object,
 - the thread is assumed to hold a lock associated with the Condition object.
 - If the thread does not, an IllegalMonitorStateException is thrown.

2-Step Thread Termination



HW 14

- Implement 2-step termination for "deposit" and "withdraw" threads.
 - Have the main thread call interrupt() on "deposit" and "withdraw" threads
 - To let those threads to wake up in case they are in the Waiting state due to await() or sleep().
- Due: April 18 (Thu) midnight

signalAll() *Before* or *After* a State Change?

```
withdraw(double amount) {
 lock.lock();
 while(balance =< 0){</pre>
     // waiting for the balance to exceed 0
    sufficientFundsCondition.await(); }
 balance -= amount;
 belowUpperLimitFundsCondition.signalAll();
 lock.unlock(); }
deposit(double amount) {
 lock.lock();
 while(balance >= 300) {
    // waiting for the balance to go below 300.
    belowUpperLimitFundsCondition.await(); }
 balance += amount;
 sufficientFundsCondition.signalAll();
 lock.unlock(); }
```

• What if you call signalAll() first and then update the balance? Will any thread safety issues come out?

```
vithdraw(double amount){
(1) W thread:
             lock.lock();
"waiting"
             while(balance <= 0){
temporarily
                  // waiting for the balance to exceed 0
releases the
                 sufficientFundsCondition.await(); }
             belowUpperLimitFundsCondition.signalAll(); 
lock
             balance -= amount;
                                                             D thread: wakes
             lock.unlock(); }
                                                             up W thread.
                                                             which is waiting
            deposit(double amount) {
                                                             until balance > 0.
             lock.lock();
             while(balance >= 300) {
                 // waiting for the balance to go below 300.
(2) D thread:
                 belowUpperLimitFundsCondition.await(); }
signalAll().
             sufficientFundsCondition.signalAll();_
Ctx switch
             balance += amount;
             lock.unlock(); }
```

- Can the "W" thread withdraw money before the "D" thread deposits money?
 - Can the balance have a negative value?
 - The answer is NO.

```
    withdraw(double amount) {

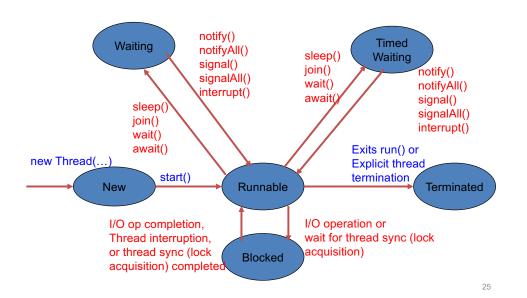
   lock.lock();
   while (balance =< 0) {
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await(); }
   belowUpperLimitFundsCondition.signalAll();
   balance -= amount;
   lock.unlock(); }
  deposit(double amount) {
   lock.lock();
   while(balance >= 300) {
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   sufficientFundsCondition.signalAll();
   balance += amount;
   lock.unlock(); }
```

• For example, do you need to worry about race conditions in this case?

```
(1) W thread: vithdraw (double amount) {
              Lock.lock();
"waiting"
              hile(balance <= 0){
                  // waiting for the balance to exceed 0(3) W thread:
temporarily
                                                           "runnable"
releases the
                 sufficientFundsCondition.await(); }
             belowUpperLimitFundsCondition.signalAll(); Tries to acquire the
lock
             balance -= amount;
                                                           lock again and fails.
             lock.unlock(); }
                                                           Goes to "blocked."
            deposit(double amount) {
             lock.lock();
             while(balance >= 300) {
                 // waiting for the balance to go below 300.
(2) D thread:
                 belowUpperLimitFundsCondition.await(); }
signalAll().
             sufficientFundsCondition.signalAll();
Ctx switch
             balance += amount;
             lock.unlock();
}
```

- "W" thread CANNOT withdraw money before "D" thread deposits money.
- "D" thread CANNOT deposit money before "W" thread withdraws money.

States of a Thread



Two Important Things (1)

- You can safely change the state/value of a shared variable after calling signalAll().
 - AS FAR AS the state changes in atomic code
- That said, common programming convention/practice is:
 - A state change first, followed by signalAll().

Two Important Things (2)

- A JVM can perform context switches even when a thread runs atomic code.
 - A lock guarantees that only one thread exclusively runs atomic code at a time.
 - Some resources (books, online materials, etc.)
 explicitly/implicitly say that context switches never occur when a thread runs atomic code.
 - It is WRONG!

signal() and signalAll()

- signalAll()
 - Wakes up all waiting threads on a condition object.
 - All of them go to the "runnable" state.
 - One of them will re-acquire a lock. The others will go to the "blocked" state.
- signal()
 - Wakes up one of waiting threads on a condition object.
 - The selected thread goes to the "runnable" state. The others stay at the "waiting" state.
 - JVM's thread scheduler selects one of them. Assume random selection.
 - Not predictable which waiting thread to be selected.

signal() and signalAll()?

- Either one works well.
- signalAll() is favored in many cases/projects.
 - I prefer signalAll() in my personal taste.

"while" or "if" to Surround await()?

```
    withdraw(double amount) {

   lock.lock();
   while(balance <= 0) {</pre>
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await(); }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }
  deposit(double amount) {
   lock.lock();
   while (balance >= 300) {
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
```

 "while" should be used rather than "if" when multiple threads call withdraw() concurrently. Why?

ThreadSafeBankAccount2.java

```
    Condition sufficientFundsCondition = lock.newCondition();

  Condition belowUpperLimitFundsCondition = lock.newCondition();

    withdraw(double amount) {

   lock.lock();
   while (balance <= 0) {
       // waiting for the balance to exceed 0
       sufficientFundsCondition.await(); }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }

    deposit(double amount) {

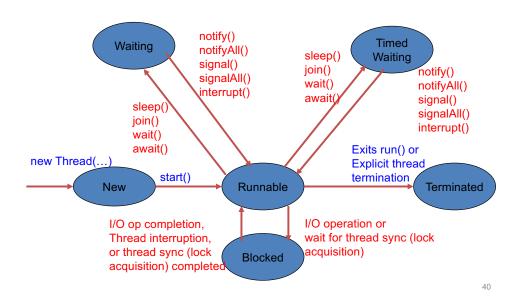
   lock.lock();
   while (balance >= 300) {
       // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
```

A Potential Problem

37

```
(3) Two W threads:
            withdraw(double amount) {
                                                       Go to "runnable" One
(1) b==0.
             lock.lock();
                                                       of them acquires the
Two W
             if(balance =< 0){
                  // waiting for the balance to excee lock again.
threads:
                 sufficientFundsCondition.await(); }
"waiting"
             balance -= amount;
                                                       The other W thread:
             belowUpperLimitFundsCondition.signalAl
                                                       Go to "blocked" on
             lock.unlock(); }
                                                       acquiring the lock.
            deposit(double amount) {
             lock.lock();
             if(balance >= 300){
                 // waiting for the balance to go below 300.
(2) D thread:
                 belowUpperLimitFundsCondition.await(); }
             balance += amount;
signalAll()
             sufficientFundsCondition.signalAll();
followed by
             lock.unlock(); }
unlock()
b = = 100
```

States of a Thread



```
them acquires the lock
                                                       again, withdraws money
                                                       and releases the lock.
             withdraw(double amount) {
(1) b==0.
                                                       b = = 0.
              lock.lock();
Two W
              if(balance =< 0){
threads:
                  // waiting for the balance to exceed 0
                 sufficientFundsCondition.await():
"waiting"
             balance -= amount;
             belowUpperLimitFundsCondition.signalAll()
             lock.unlock(); }
                                                         (4) The 2nd W thread
                                                          acquires the lock and
            deposit(double amount) {
                                                         withdraws money.
             lock.lock();
                                                         b==-100.
             if(balance >= 300) {
                 // waiting for the balance to go below 300.
(2) D thread:
                 belowUpperLimitFundsCondition.await(); }
signalAll()
             balance += amount:
followed by sufficientFundsCondition.signalAll();
             lock.unlock(); }
unlock()
b = = 100
```

(3) Two W threads:

Go to "runnable" One of

```
(3) Two W threads:
              ithdraw(double amount) {
                                                        "runnable" One of them
(1) b=0.
              lock.lock();
                                                        acquires the lock again
Two W
              f(balance = < 0)
                                                        and releases it.
threads:
                  // waiting for the balance to excee
                 sufficientFundsCondition.await()_
                                                        balance==0.
"waiting"
             balance -= amount:
                                                        The other W thread:
             belowUpperLimitFundsCondition.signalAl
                                                        "blocked" on acquiring
             lock.unlock(); }
                                                        the lock.
            deposit(double amount) {
                                                          (4) The 2nd W thread
             lock.lock();
                                                          acquires the lock and
             if(balance >= 300) {
                 // waiting for the balance to go below releases it.
(2) D thread:
                 belowUpperLimitFundsCondition.await(); b=-100.
signalAll()
             balance += amount;
followed by sufficientFundsCondition.signalAll();
             lock.unlock(); }
unlock()
balance==100
```

- The 2nd "W" thread should have made sure if balance>0.
- If only one "W" thread runs, this problem does not occur.
- Just always use a while loop regardless of the number of threads you use.

"if" or "while" in Atomic Code?

- You can use "if", rather than "while," for a conditional check
 - if you use signal(), not signalAll().
- However, in practice, the **while-signalAll** pair is more common than the **if-signal** pair.