



What is a Thread?

- A thread is a sequential flow of control within a program.
 - Each step in a thread (assignment, arithmetic/logical computation, conditional test, function call) must run to completion before the following step can begin.
 - Dependencies in algorithms (where computational steps rely on the results of previous steps, and so must begin after the prior step finishes) are often implicit in the sequence of steps in a program. Programmers learn to order program steps so as to satisfy all the dependencies in an algorithm.
-

What is a Thread?

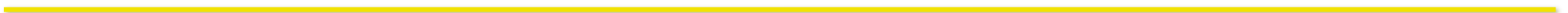
- Just because one program step follows another doesn't mean a dependency exists: the steps may be independent, and the programmer had to choose (maybe arbitrarily) which one to put first.
 - In many cases, the algorithm would work correctly with multiple steps executing simultaneously. This is called parallelism.
-

What is a Thread?

- Modern computers and operating systems have the ability to run multiple programs simultaneously (multitasking). This is useful because:
 - If one program pauses while a computation completes, a document opens, or a web page loads, the user can switch to another program and continue working in the meantime.
 - If a program only acts at certain time intervals (checking for email, scanning for viruses, etc.), it can linger in memory rather than require manual activation by the user.
 - If a computer provides service to multiple users, it can serve them at the same time (up to a capacity limit).
-

What is a Thread?

- If a computer has multiple CPUs, the operating system will put each program on a different CPU.
 - Many CPUs now have multiple computational “cores,” each of which can be executing a separate program. Each core looks like a distinct CPU to the operating system.



What is a Thread?

- The multitasking concept has been extended such that individual programs can have multiple threads of control (multithreading).
 - Each of these “sub-programs” or threads executes independently of and in parallel with the other threads in the program.
 - Within each thread, steps are still processed sequentially.
 - Each thread has its own private call stack to keep track of its independent Program Counter, local variables, method arguments, etc.
 - Classes, objects, member variables, and static variables are shared by all threads in a program.
-

Creating Threads

- You can create new threads by either
 1. Subclassing `java.lang.Thread`, instantiating it, and calling its `start()` method.
 2. Implementing interface `java.lang.Runnable` in a class, instantiating a `Thread` with the `Runnable` as an argument, and calling `start()` on the `Thread`.
 - `Thread` and `Runnable` both define a `run()` method which gets called when the new thread starts.
 - The `start()` method of `Thread` returns immediately, so the method that called `start()` will be running in parallel with the `run()` method of the new thread.
 - There are no hard limits to the number of threads you may create or have running at any time.
 - Practical limits depend on the platform.
-

How to Create a Thread (Method 1)

```
public class Counter extends Thread // Subclassing java.lang.Thread
{
    public Counter()
    {
        super("Counter"); // Gives the thread a name
    }

    /**
     * Thread.run() is called when the thread starts. When this method exits,
     * the thread dies.
     */
    public void run()
    {
        for (int i = 0; i < 100; i++)
            System.out.println(i);
    }

    /**
     * The main thread (created by the JVM) calls this when it starts. When main()
     * exits, the main thread dies. If no other threads were created, the JVM
     * exits when main() exits.
     */
    public static void main(String[] args)
    {
        Counter c = new Counter();
        c.start(); // New thread will be scheduled to run by the OS, main thread dies
    }
}
```

How to Create a Thread (Method 2)

```
public class Counter implements Runnable
{
    /**
     * This is the only method declared by the java.lang.Runnable interface.
     */
    public void run()
    {
        for (int i = 0; i < 100; i++)
            System.out.println(i);
    }
}

public class SomeOtherClass
{
    public static void main(String[] args)
    {
        Counter c = new Counter();
        Thread t = new Thread(c); // Thread constructor can take a Runnable argument
        t.start(); // interface method Runnable.run() will be called Thread starts
    }
}
```

Threads are Good

- Threads are useful because:
 - Increases in computing performance now come mostly in the form of more processing units instead of faster units.
 - A single thread can only run on one CPU at a time because of the “state” information that needs to be maintained in the CPU (program counter, local variables, etc.)
 - A multithreaded program can continue responding to user actions while running a long computation, manipulating a large file, communicating with a printer, etc.
 - If a thread gets stuck waiting for disk I/O, other threads can continue using other available resources (network I/O, USB I/O, graphics, etc.).
-

Threads are Evil

- Threads are the bane of human existence because:
 - Splitting algorithms into multiple threads (parallelizing) can be hard. Some problems don't divide easily into independent sub-problems.
 - Threads cannot be coordinated by function call semantics; communicating between threads must be done using more complex mechanisms.
 - Steps will execute in a different order when a program is run on a different configuration of computer. In fact, differences in "background activity" (what other programs and the OS are doing) will cause each run of a program to be different.
 - If multiple threads try to access shared memory at the same time, chaos results.
-

Preemptive vs. Cooperative Multitasking

- If there are more active threads than CPUs/cores, multiple threads have to share each CPU.
- If the CPU(s) rotate quickly through the threads, working on each for a small “slice” of time before rotating to the next thread, the illusion is maintained of each thread having its own CPU. There is no loss of responsiveness.
- In the early days of multitasking, programs would tell the operating system when it was “safe” to switch tasks (i.e., the program wasn’t in the middle of changing some important data structure, which was in a momentarily inconsistent state).

Preemptive vs. Cooperative Multitasking

- In the early days of multitasking, programs would tell the operating system when it was “safe” to switch tasks (i.e., the program wasn’t in the middle of changing some important data structure, which was in a momentarily inconsistent state).
 - Writing code to frequently “wrap up” what the program is doing and transfer control to another program is tedious.
 - If a program failed to do this, all other programs were frozen until it “yielded” the CPU.
-

Preemptive vs. Cooperative Multitasking

- Contemporary operating systems don't rely on programs to yield control. When the OS determines it's time for a thread to give up the CPU, it transfers control automatically.
 - When a thread gets to execute again, the OS restores the call stack (including Program Counter) to the same condition it was in at preemption time.
 - The thread has no indication that it was interrupted, but other threads may have changed shared objects during the interruption.
 - Preemption can occur on any program step.
 - Two threads accessing the same data can encounter a race condition, where the OS task switches when one thread is actively modifying the data, potentially confusing the other thread.
-

Race Condition

The increment operator:

```
x++;
```

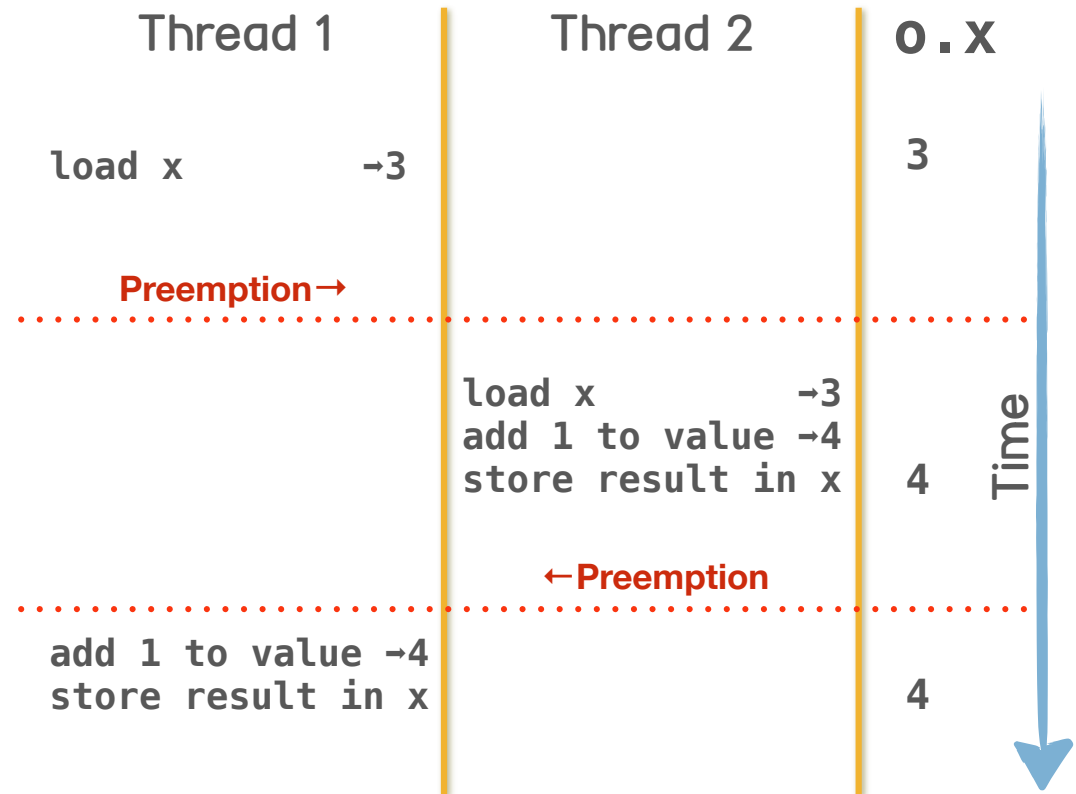
Is a compiler shorthand for:

```
x = x + 1;
```

Which is implemented as 3 CPU operations:

```
load x  
add 1 to x  
store result in x
```

If two threads both try to execute `o.x++` on shared object `o`, the following might result:



Critical Sections

- To make up for the lack of control when a thread switch occurs, preemptive multitasking systems let you “lock” pieces of data to prevent simultaneous access by other threads.
 - Sections of code which cannot safely be executed by more than one thread at a time (because they modify shared data) are called critical sections.
 - You mark a critical section using a **synchronized** block.
 - You can mark an entire method as a critical section by using the **synchronized** modifier.
-

Critical Sections

- When a thread reaches a **synchronized** block, it will acquire the monitor (a kind of lock mechanism) for the object.
 - If another object already has the monitor, the current thread will block until the monitor becomes available.
 - Once the current thread succeeds in getting the monitor, it will enter the critical section.
 - When it finishes the critical section, it will release the monitor and give other threads a chance to acquire it.
-

Critical Sections

```
public class MyThreadSafeCollection
{
    private int size;
    private Object[] data;

    // Acquires the monitor on 'this' instance
    public synchronized void size()
    {
        return this.size;
    }

    public void add(int index, Object element)
    {
        synchronized (this) // Acquires the monitor named in parenthesis
        {
            System.arraycopy(this.data, index, this.data, index + 1,
                this.size - index - 1);
            this.data[index] = element;
            this.size++;
        }
    }
}
```

Both approaches have the same behavior here, but the second lets you control of the scope of the critical section.

Synchronized Data Structures

- We recently talked about unmodifiability wrappers, which can give us a read-only view into a collection.
- There are also synchronized wrappers, which give us a “thread-safe” view into a collection, so that multiple threads can access the same collection without corrupting it. These are also in `java.util.Collections`:
 - `Collection c = Collections.synchronizedCollection(coll);`
 - `List l = Collections.synchronizedList(list);`
 - `Set s = Collections.synchronizedSet(set);`
 - `SortedSet s = Collections.synchronizedSortedSet(ss);`
 - `Map m = Collections.synchronizedMap(map);`
 - `SortedMap m = Collections.synchronizedSortedMap(map);`
- Each method in the wrapper locks the collection before accessing or modifying it.
- Locking before read access is required because of cache-consistency issues in multiple processor environments.

Inter-Thread Communication

- Methods in a single-threaded environment communicate by method arguments and return values. This mechanism isn't available to multi-threaded applications because the caller and recipient seldom coincide at the same method invocation.
 - A mechanism is needed which brings multiple threads to the same execution point so they are ready to exchange data.
- `wait()` and `notify()` are methods on `java.lang.Object` (so they are inherited by every object) that provide a means to do this.

Inter-Thread Communication

- Calling `wait()` causes the current Thread to stop executing until another Thread calls `notify()` on the same Object.
 - `wait()` and `notify()` can only be called from inside a `synchronized` block whose target is the same object as the object whose `wait()` or `notify()` method is being called.
 - If you are calling `x.wait()`, you have to be inside a `synchronized(x)` block.
-

Inter-Thread Communication

- When a **Thread** enters a `wait()`, it temporarily releases the monitor that it was holding as part of the **synchronized** block.
 - When a **Thread** is wakened from a `wait()`, it must reacquire the monitor before resuming execution.
 - Each **Object** can maintain a “list” of **Threads** waiting on it. The `notify()` method wakes one of those **Threads**. Another method, `notifyAll()` wakes them all.
 - If no **Threads** are waiting on an **Object**, `notify()` and `notifyAll()` do nothing.
-

Using wait()/notify() to Exchange Data

```
public class MyClass
{
    private List<Object> l;

    public synchronized Object getNextThing()
    {
        while (this.l.size() == 0) // Wait until there is data available
        {
            try
            {
                this.wait(); // Thread sleeps until some other Thread calls notify()
            }
            catch (InterruptedException ie) { }
        }
        return this.l.remove(0);
    }

    public synchronized void putNextThing(Object o)
    {
        this.l.add(o);
        this.notify(); // Will "wake" thread waiting for data, if there is any
    }
}
```

In-class lab exercise #16

Producer/Consumer

Create a program that uses multiple threads to implement a producer/consumer. The producer thread generates numbers for some other thread to consume. The consumer thread waits until the producer thread has produced something, and then consumes it.

16

Producer/Consumer

Create a producer/consumer:

1. Create a producer **Thread** subclass. The producer **Thread** has a **List** containing the produced elements.
 - 1.1. Write a **get()** method that removes and returns the first element from the list. If there are no elements, do a **wait()**. This method will be called by the **Consumer**.
 - 1.2. Write a **run()** method that produces three values (you can use a counter to generate different values), adds them to the list, send **notify()** to the consumer, then sleeps for **3** seconds before repeating the process.
 - 1.3. Ensure only one **Thread** at a time accesses the **List**.
2. Create a consumer **Thread** subclass. Each **Consumer** instance needs a reference to a **Producer** instance.
 - 2.1. Write a **run()** method that consumes one value, prints it to the screen, then sleeps for **500** milliseconds. (It will consume faster than the producer produces.)
3. Driver: Create a **Producer** and a **Consumer** instance, giving the **Consumer** a reference to the **Producer**.
 - 3.1. Start both threads.

Using Multiple Locks

```
public class BankAccount
{
    private int number;
    /** Current balance. All access to this variable is controlled by the monitor
     *  on the enclosing BankAccount instance */
    private double balance;

    public int getNumber()
    {
        return this.number;
    }

    public synchronized double getBalance()
    {
        return this.balance;
    }

    public synchronized void setBalance(double balance)
    {
        this.balance = balance;
    }
}
```

Using Multiple Locks

```
public class BankAccount
{
    /**
     * This method is static because it affects two instances.
     */
    public static transfer(Account fromAccount, Account toAccount, double amount)
        throws OverdraftException
    {
        if (amount <= 0.0)
            throw new IllegalArgumentException("Transfer amount must be positive.");
        double fromBalance = fromAccount.getBalance();
        double newBalance = fromBalance - amount;
        if (newBalance < 0.0)
            throw new OverdraftException(newBalance);
        fromAccount.setBalance(newBalance);
        toAccount.setBalance(toAccount.getBalance() + amount);
    }
}
```

Using Multiple Locks

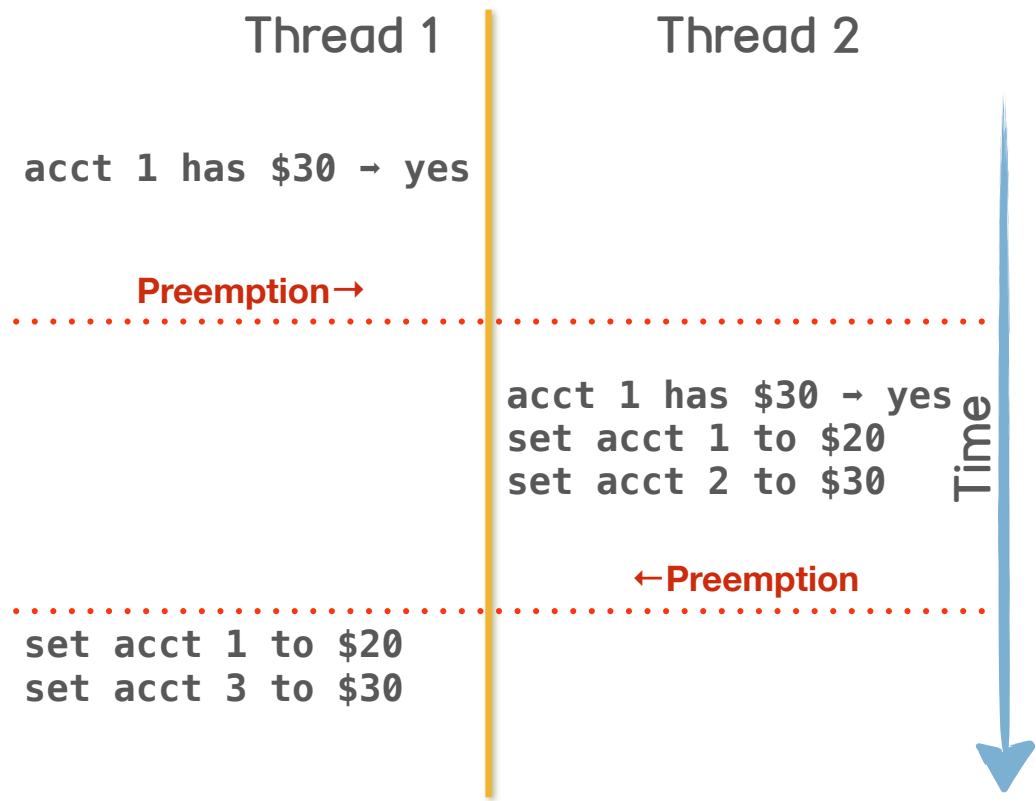
Suppose account #1 has \$50 and a transfer of \$30 from account #1 to #2 is initiated at the same time as a transfer of \$30 from account #1 to account #3. Assume accounts #2 and #3 start with \$0 balances.



Using Multiple Locks

The `transfer()` method has a race condition. It performs 3 operations:

1. Check sufficient balance in 1st account
2. Deduct money from 1st account
3. Add money to 2nd account



Before the operations, the accounts has a total balance of \$50. After the operations, the accounts have a total balance of \$80. The program has created bank obligations without deposits.


Using Multiple Locks

Making the accessors and mutators **synchronized** prevents corruption of a single object, but it doesn't help coordinate multiple objects. The static `transfer()` method also acquires and releases monitors as it enters and exits **synchronized** instance members, leaving gaps where another thread can alter the object. The static method needs to hold the monitors for both objects for the entire time it's executing...



Using Multiple Locks

```
public class BankAccount
{
    /**
     * This method is static because it affects two instances.
     */
    public static transfer(Account fromAccount, Account toAccount, double amount)
        throws OverdraftException
    {
        if (amount <= 0.0)
            throw new IllegalArgumentException("Transfer amount must be positive.");
        synchronized (fromAccount)
        {
            synchronized (toAccount)
            {
                double fromBalance = fromAccount.getBalance();
                double newBalance = fromBalance - amount;
                if (newBalance < 0.0)
                    throw new OverdraftException(newBalance);
                fromAccount.setBalance(newBalance);
                toAccount.setBalance(toAccount.getBalance() + amount);
            }
        }
    }
}
```

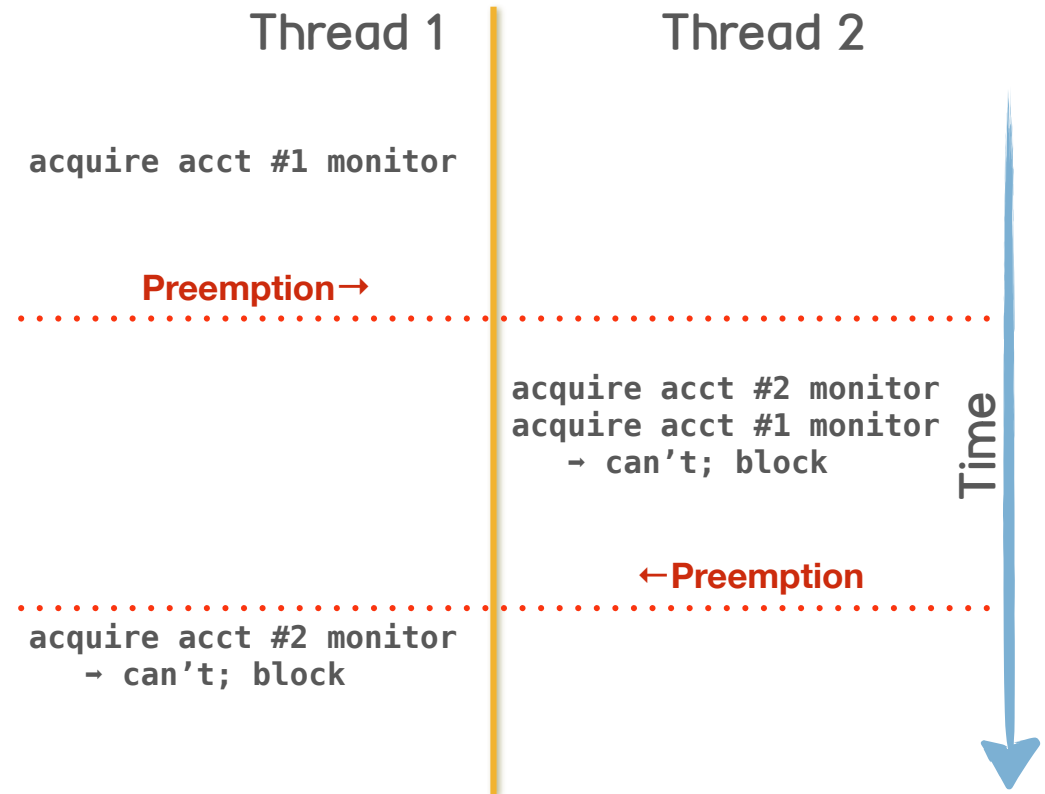


synchronized method called from
inside the synchronized block

A thread that already holds a monitor can reacquire the same monitor multiple times

Using Multiple Locks

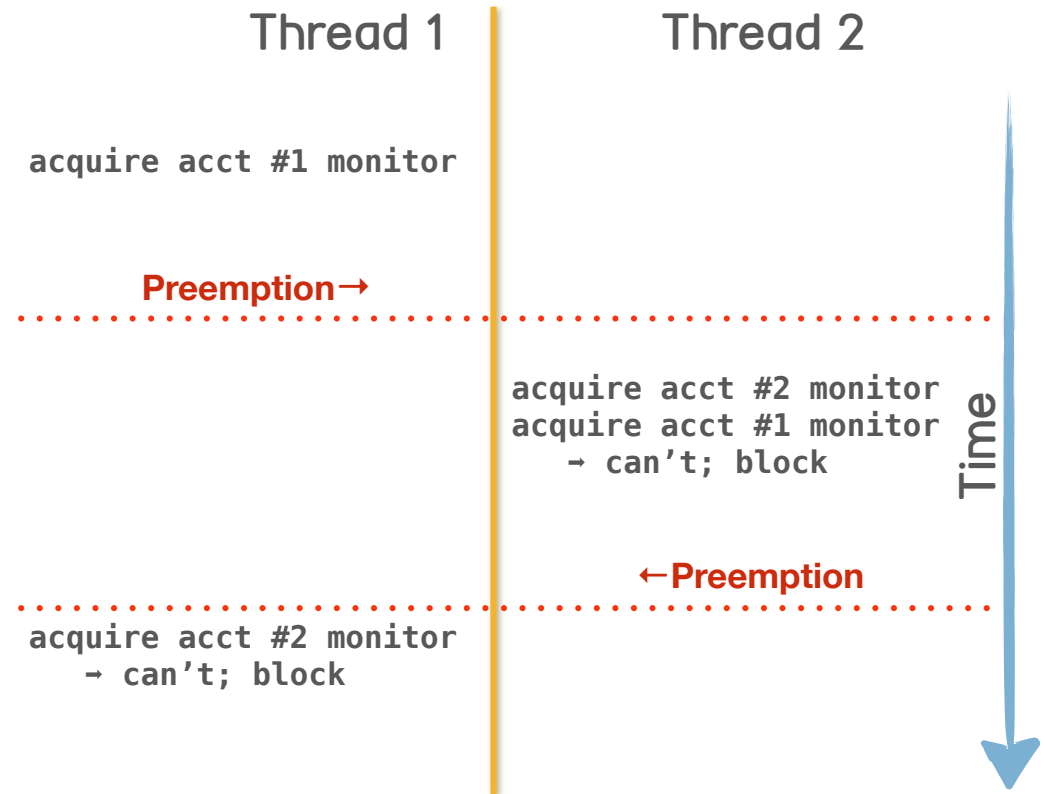
This solution creates a worse problem. Consider a case when a transfer from account #1 to account #2 happens at the same time as a transfer from account #2 to account #1:



Thread 1 is holding the Account 1 monitor, waiting for Thread 2 to give up the Account 2 monitor. Thread 2 is holding the Account 2 monitor, waiting for Thread 1 to give up the Account 1 monitor.

Using Multiple Locks

This solution creates a worse problem. Consider a case when a transfer from account #1 to account #2 happens at the same time as a transfer from account #2 to account #1:



This situation has no way of resolving itself, and both threads (and both accounts) will be “frozen” until the program is terminated. Any other thread that tries to access either account will be frozen also.

Deadlocks

- Deadlocks are difficult to debug.
 - Their manifestation is highly timing dependent.
 - The method calls that acquired the monitors could be deep in the call stack.
 - No errors are generated when deadlocks occur.
 - Deadlocks are identified by the JVM thread trace mechanism, which will identify the threads and objects involved.
 - Press Control-Break from a Windows shell
 - Execute “`kill -QUIT {pid}`” from a Unix shell
 - Careful design, including analysis of possible control-flow paths is necessary to avoid deadlocks.
 - One technique is to always acquire monitors in a given order.
-

Deadlocks

```
public class BankAccount
{
    public static transfer(Account fromAccount, Account toAccount, double amount)
        throws OverdraftException
    {
        if (amount <= 0.0)
            throw new IllegalArgumentException("Transfer amount must be positive.");
        Account first, second;
        if (fromAccount.getNumber() < toAccount.getNumber())
            { first = fromAccount; second = toAccount; }
        else
            { first = toAccount; second = fromAccount; }
        synchronized (first)
        {
            synchronized (second)
            {
                double fromBalance = fromAccount.getBalance();
                double newBalance = fromBalance - amount;
                if (newBalance < 0.0)
                    throw new OverdraftException(newBalance);
                fromAccount.setBalance(newBalance);
                toAccount.setBalance(toAccount.getBalance() + amount);
            }
        }
    }
}
```

Alternate Approach

```
public class BankAccount
{
    private int number;
    /** Current balance. All access to this variable is controlled by the monitor
     *  on the enclosing BankAccount instance */
    private double balance;

    public synchronized double getBalance()
    {
        return this.balance;
    }

    /**
     * Alter the account balance by a given amount.
     * @param change Positive value represents a contribution to a balance.
     * Negative value reflects a deduction.
     * @return New balance
     */
    public synchronized double transaction(double change) throws OverdraftException
    {
        double newBalance = this.balance + change;
        if (newBalance < 0.0)
            throw new OverdraftException(newBalance);
        this.balance = newBalance;
        return this.balance;
    }
}
```

Alternate Approach

```
public class BankAccount
{
    /**
     * This method is static because it affects two instances.
     */
    public static transfer(Account fromAccount, Account toAccount, double amount)
        throws OverdraftException
    {
        if (amount <= 0.0)
            throw new IllegalArgumentException("Transfer amount must be positive.");
        fromAccount.transaction(-amount);
        toAccount.transaction(amount);
    }
}
```

Threads and GUIs

- The AWT maintains a separate thread for all GUI-related functions: event delivery, screen repainting, etc.
 - There is only one event thread to avoid dealing with synchronization issues in GUI code.
 - The assumption is GUIs are programmed by graphic artists/non-CS types who would find multi-threading too complex.
 - The CS types would be assigned server-side work where multi-user requirements require advanced knowledge of thread synchronization.
-

Threads and GUIs

- Any long computation inside an event listener or paint method prevents other events from being dispatched. This causes all painting to freeze and all GUI widgets to become non-responsive.
 - Any method that could take a long time to execute (computation, I/O, etc.) must be done on a thread other than the event thread.
 - But, because AWT/JFC are not “thread safe,” all modifications to windows, GUI widgets, etc. must be done on the event thread.
-

Threads and GUIs

- To perform a time-consuming operation based on a GUI action:
 1. Have the listener start a new thread, then return.
 2. Have the new thread perform the operation.
 3. Update the GUI to indicate the operation is complete from the event thread. (How?)
- If another thread needs to get the event thread to execute some code, it:
 - Defines a `Runnable` class (Lecture 25, pg. 7) that contains the desired behavior.
 - Instantiates an instance
 - Passes the instance to
`javax.swing.SwingUtilities.invokeLater(runnable)`
- The `Runnable` will then be treated like a GUI event and dispatched through the event queue.

In-class lab exercise #17

Threads & GUIs

Modify a program that does not use threads properly to avoid monopolizing the (only) system event thread.

17

Threads & GUIs

Improve the threading behavior of a class:

1. Download the application class **ThreadTest.java** from the web site.
2. Observe how clicking either button freezes the animation.
3. Create a new thread class to process the call to **button2Work()**.
4. Change **button2()** to create the new thread instead of calling **button2Work()** directly from the event thread.
5. Have the new thread finish by kicking off a **Runnable** that restores Button 2 to enabled status.