CSx55: Distributed Systems [Threads]

Threads block when they can't get that lock

Wanna have your threads stall?
    Go ahead, synchronize it all

The antidote to this liveness pitfall?
    Keeping the lock scope small

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Is there a max nesting depth for calls?
- Who determines which task will run where there is a memory stall?
- Within an application does it ever make sense to use IPC instead of threads?
- Can page faults occur for thread creation?
- Is there a max number of threads per process?
- Can threads control/coordinate with other threads within a process?
- Multiple threads with multiple cores? Cores with multiple ALUs?

Topics covered in this lecture

- Threads
  - Thread Lifecycle
- Data synchronization
- Synchronized blocks

How big a stack? [1/2]

- The size of the stack must be large enough to accommodate the **deepest nesting level** needed during the thread's *lifetime*
- Kernel threads
  - Kernel stacks are allocated in physical memory
  - The nesting depth for kernel threads tends to be small
  - E.g., 8KB default in Linux on an Intel x86
  - Buffers and data structures are allocated on the heap and never as procedure local variables

How big a stack? [2/2]

- User-level stacks are allocated in virtual memory
- To catch program errors
  - Most OS will trigger **error** if the program stack grows too large too quickly
    - Indication of an unbounded recursion
  - Google's GO will automatically grow the stack as needed … this is very uncommon
  - POSIX, for e.g., allows default stack size to be library dependent (e.g., larger on a desktop, smaller on a phone)
    - "Exceeding default stack limit is very easy to do, with the usual results"
      - Program termination

Thread Lifecycle

Lifecycle of a thread

- Creation
- Starting
- Terminating
- Pausing, suspending, and resuming

Thread: Methods that impact the thread's lifecycle

```
public class Thread implements Runnable {
    public void start();
    public void run();
    public void stop();
    public void resume();
    public void suspend();
    public static void sleep(long millis);
    public boolean isAlive();
    public void interrupt();
    public boolean isInterrupted();
    public static boolean interrupted();
    public void join();
}
```

Deprecated, do not use (stop(), resume(), suspend())

Thread creation

- Threads are represented by instances of the `Thread` class
- When you extend the `Thread` class?
  - Your instances are also `Threads`
- We looked at the 4 constructor arguments in the `Thread` class

Starting a thread [1/2]

- Thread exists once it's been constructed
  - But it is *not executing* … it's in a **waiting** state
- In the waiting state, other threads can *interact* with the existing **thread object**
  - Object state may be changed by other threads
    - Via method invocations

Starting a thread [2/2]

- When we're ready for a thread to begin executing code
  - Call the **`start()`** method
  - `start()` performs internal house-keeping and *then calls* the **`run()`** method

- When the `start()` method returns?
  - **Two threads** are executing in parallel
    1. The original thread which just returned from calling **`start()`**
    2. The newly started thread that is executing its **`run()`** method

After a thread's `start()` method is called

- The new thread is said to be **alive**
- The `isAlive()` method tells you about the state
  - `true`: Thread has been started and *is executing* its `run()` method
  - `false`: Thread may *not be started* yet or may be *terminated*

Terminating a thread

- Once started, a thread executes only one method: `run()`
- This `run()` may be complicated
  - May *execute forever*
  - Call *several other methods*
- Once the `run()` *finishes* executing, the thread has **completed** its execution

Like all Java methods, `run()` finishes when it …

① Executes a `return` statement

② Executes the last statement in its method body

③ When it *throws an exception*

- Or fails to catch an exception thrown *to it*

The only way to terminate a thread?

- Arrange for its `run()` method to **complete**
- But the documentation for the `Thread` class lists a `stop()` method?
  - This has a *race condition* (unsafe), and has been deprecated

Some more about the `run()` method

- Cannot throw a ***checked*** exception
- But it can throw an ***unchecked*** exception
  - Exception that extends the `RuntimeException`
- A thread can be **stopped** by:
  1. *Throwing* an unchecked exception in `run()`
  2. *Failing to catch* an unchecked exception thrown by something that `run()` has called

Pausing, suspending and resuming threads

- Some thread models support the concept of **thread suspension**
  - Thread is told to *pause* execution and then told to *resume* its execution
- Thread contains `suspend()` and `resume()`
  - Suffers from vulnerability to *race conditions*: **deprecated**
- Thread can *suspend its own execution* for a specified period
  - By calling the `sleep()` method

But sleeping is not the same thing as thread suspension

- With true thread suspension
  - One thread can suspend (and later resume) *another thread*
- `sleep()` affects only the thread that executes it
  - Not possible to tell another thread to go to sleep

But you can achieve the functionality of suspension and resumption

- Use `wait` and `notify` mechanisms

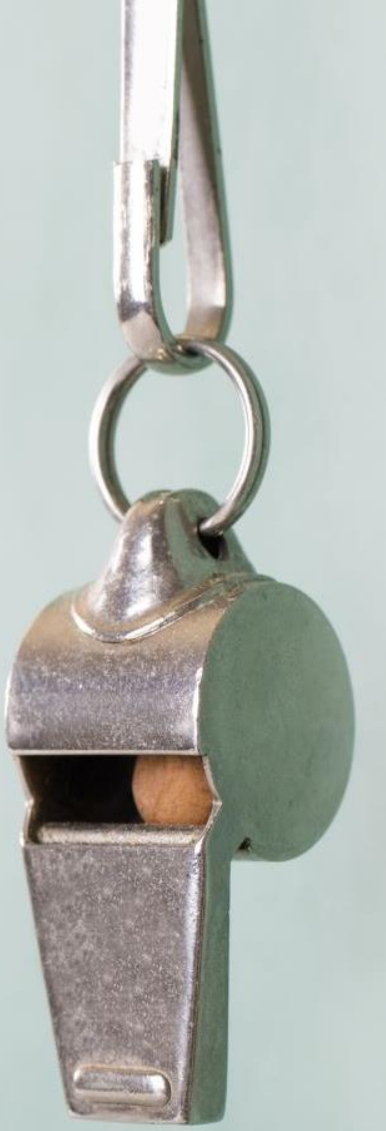
- Threads **must be coded** to use this technique
  - This is not a generic suspend/resume that is imposed by another thread

Thread cleanup

- As long as some other active object holds a reference to the terminated thread object
  - Other threads can execute methods on the terminated thread … retrieve information
- If the object representing the terminated thread goes *out of scope*?
  - The thread object is **garbage collected**

Holding onto a thread reference allows us to determine if work was completed

- Done using the `join()` method
- The `join()` method
  - **Blocks** until the thread has completed
  - *Returns immediately* if
    - The thread has already completed its `run()` method
      - You can call `join()` any number of times
- Don't use `join()` to poll if the thread is still running
  - Use `isAlive()`



Stopping a thread

Two approaches to stopping a thread

- Setting a flag
- Interrupting a thread

Stopping a Thread: Setting a flag

- **Set some internal flag** to signal that the thread should stop
- Thread periodically **queries the flag** to determine if it should exit

Stopping a Thread: Setting a flag

```
public class RandomGen extends Thread {
   private volatile boolean done = false;

   public void run() {
     while (!done) {
         ...
     }
   }

   public void setDone() {
      done = true;
   }
}
```

`run()` method investigates the state of the `done` variable on every loop. Returns when the `done` flag has been set.

Interrupting a thread

- In the previous slide, there may be a *delay* in the `setDone()` being invoked & thread terminating
  - Some statements are executed after `setDone()` and before the value of `done` is checked
  - In the worst case, `setDone()` is called right after the the `done` variable was checked
- **Delays** while waiting for a thread to terminate are *inevitable*
  - But it would be good if they could be minimized

Interrupting a thread

- When we arrange for thread to terminate, we:
  - Want it to *complete its blocking method* immediately
  - Don’t wish to wait for the data (or …) because the thread will exit

- Use `interrupt()` method of the `Thread` class to **interrupt** any *blocking method*

Effects of the interrupt method

- Causes blocked method to **throw** an **`InterruptedException`**
  - `sleep()`, `wait()`, `join()`, and methods to read I/O
- Sets a **flag** inside the thread object to indicate that the thread has been interrupted
  - Queried using `isInterrupted()`
    - Returns `true` if it was interrupted, even though it was not blocked

Stopping a thread: Using interrupts

```
public class RandomGen extends Thread {

   public void run() {
     while (!isInterrupted()) {
         ...
     }
   }

}
```

radomGeneratorThread.interrupt()

The `Runnable` interface

- Allows **separation** of the *implementation* of the task *from the thread* used to run task

```
public interface Runnable {

   public void run();

}
```

Creation of a thread using the `Runnable` interface

- Construct the thread
  - Pass runnable object to the thread's constructor
- Start the thread
  - Instead of starting the runnable object

Creation of a thread using the `Runnable` interface

```
public class RandomGenerator implements Runnable {

   public void run() { ... }

}

...
     generator = new RandomGenerator();
     Thread createdThread = new Thread(generator);
     createdThread.start();
```

When to use `Runnable` and `Thread`

- If you would like your class to inherit behavior from the `Thread` class
  - **Extend** `Thread`
- If your class needs to inherit from other classes
  - **Implement** `Runnable`

If you extend the `Thread` class?

- You **inherit** *behavior* and *methods* of the Thread class
  - The `interrupt()` method is part of the `Thread` class
  - You can `interrupt()` *if you extend*

Advantages of using the `Runnable` interface

- Java provides several classes that handle threading *for* you
  - Implement pooling, scheduling, or timing
  - These require the **`Runnable`** interface

But what if I still can't decide?

- Do a UML (Unified Modeling Language) model of your application
- The object hierarchy tells you what you need:
  - If your task needs to subclass another class?
    - Use `Runnable`
  - If you need to use methods of `Thread` within your class?
    - Use `Thread`

Threads and Objects

- Instance of the `Thread` class is just an **object**
  - Can be passed to other methods
  - If a thread has a reference to another thread
    - It can invoke *any method* of that thread's object
- The `Thread` object is <u>not the thread itself</u>
  - It is the set of methods and data that *encapsulate* information about the thread

But what does this mean?

- You <u>cannot</u> look at the object source and <u>know</u> *which thread is*:
  - Executing its methods or examining its data

- You may wonder about which thread is running the code, but …
  - There may be many possibilities

Determining the current thread

- Code within a thread object might want to see that code is being executed either:
  - By thread represented by the object or
  - By a completely different thread

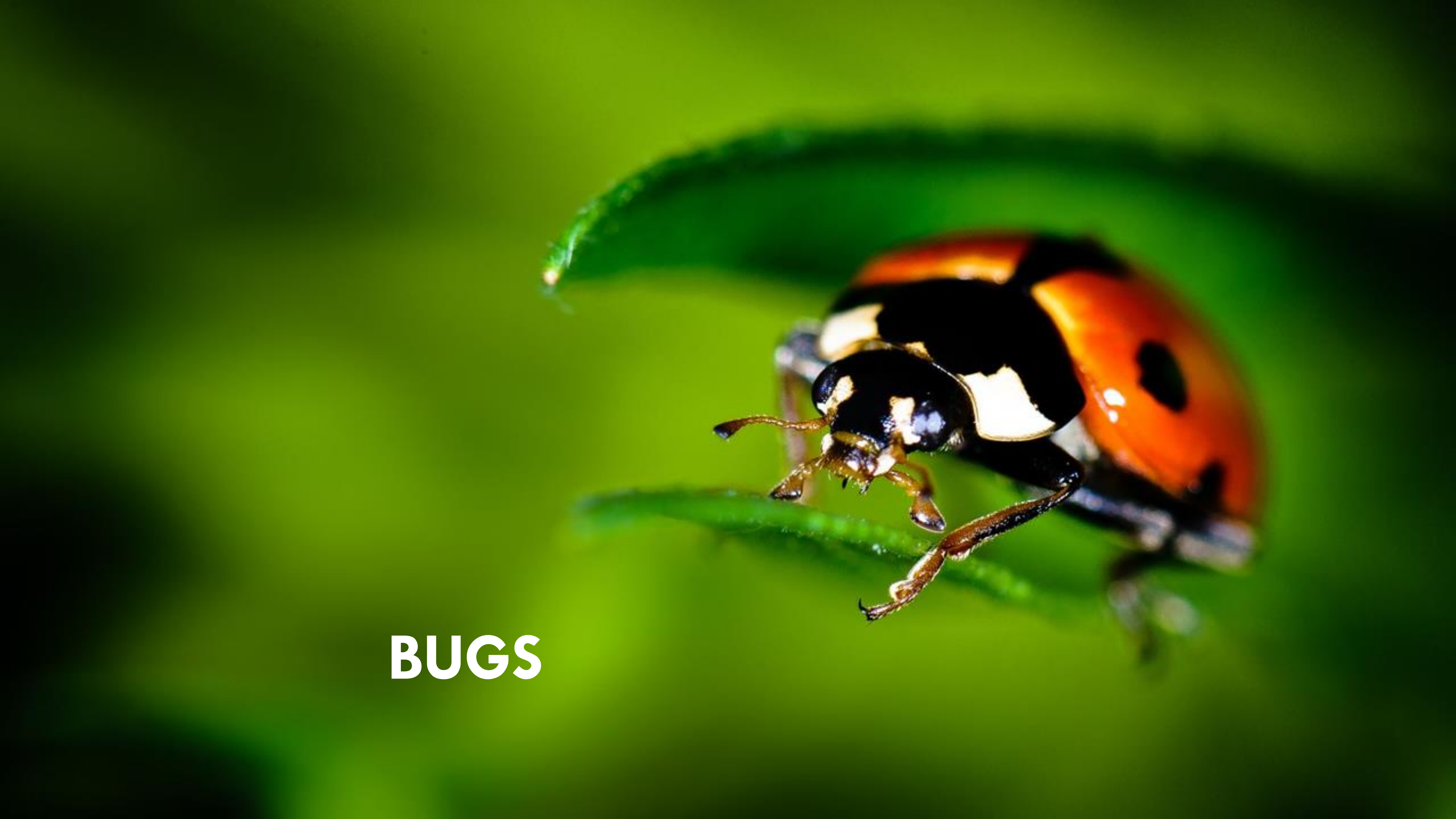
- Retrieve reference to current thread
  - `Thread.currentThread()`
  - Static method

Checking which thread is executing the code

```
public class MyThread extends Thread {

   public void run() {
      if (Thread.currentThread() != this) {
         throw new IllegalStateException
            ("Run method called by incorrect thread …);
      } /* end if */

      ... Main logic
   }

}
```

Allowing a `Runnable` object to see if it has been interrupted

```
public class MyRunnable implements Runnable {

   public void run() {
      if (!Thread.currentThread().isInterrupted() ) {
            ... Main logic
      }
   }

}
```



BUGS

Heisenbugs

- Term coined by ACM Turing Award winner Jim Gray
  - Pun on the name of Werner Heisenberg
  - Act of observing a system, alters its state!
- Describes a particular class of bugs
  - Those that disappear or change behavior when you try to examine them
- Multithreaded programs are a common source of Heisenbugs

What about regular bugs?

- Sometimes referred to as Bohr bugs
  - Deterministic
  - Generally, much easier to diagnose

Two friends plan to meet at Starbucks
But there are two Starbucks on College Avenue

	@ the First Starbucks Store	@ the Second Starbucks Store
12:10	**A** is looking for friend **B**	**B** is looking for friend **A**
12:15	**A** leaves for the second store	**B** leaves for the first store
12:20	**B** arrives at store	**A** arrives at store
12:30	**B** is Looking for friend **A**	**A** is looking for friend **B**
12:40	**B** leaves for the second store	**A** leaves for the first store

Both friends are now frustrated and undercaffeinated!

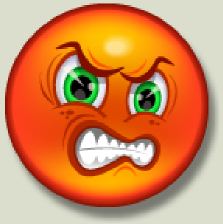
DATA SYNCHRONIZATION

Why sharing data between threads is problematic

- **Race conditions**
  - Correct outcome depends on lucky timing of uncontrollable events
- Threads attempt to access data more or less *simultaneously*
  - A thread may change the value of data that some other thread is operating on

Example code with race condition

```
public class MyThread extends Thread {
   private byte[] values;
   private int position;

   public void
      modifyData(byte[] newValues, int newPosition) {
      ... Modify values and position
   }

   public void utilizeDataAndPerformFunction() {
      ... Use values and position
   }

   public void run() {
     ... Main logic
   }
}
```



In the previous snippet a race condition exists because …

- The thread that calls `modifyData()` is **accessing the same data** as the thread that calls `utilizeDataAndPerformFunction()`

- `utilizeDataAndPerformFunction()` and `modifyData()` **are not atomic**
  - It is possible that `values` and `position` are changed *while they are being used*

What is atomic?

- The code cannot be interrupted during its execution
  - Accomplished in hardware or *simulated* in software

- Code that cannot be found in an *intermediate state*

Eliminating the race condition using the `synchronized` keyword

- If we declared both `modifyData()` and `utilizeDataAndPerformFunction()` as **synchronized**?
  - Only one thread gets to call *either* method at a time
    - Only one thread accesses data at a time
  - When one thread calls one of these methods, while another is executing one of them?
    - The second thread must *wait*

Example code with no race conditions by using the synchronized keyword

```
public class MyThread extends Thread {
    private byte[] values;
    private int position;

    public void synchronized
        modifyData(byte[] newValues, int newPosition) {
        ... Modify values and position
    }

    public void synchronized
        utilizeDataAndPerformFunction() {
        ... Use values and position
    }

    public void run() {
       ... Main logic
    }
}
```

Revisiting the mutex lock

- **Mut**ually **ex**clusive lock
- If two threads try to grab a mutex?
  - Only one succeeds
- In Java, every object has an associated **lock**

When a method is declared `synchronized` ...

- The thread that wants to execute the method must **acquire** a lock
- Once the thread has acquired the lock?
  - It executes method and **releases** the lock
- When a method returns, the lock is released
  - Even if the return is because of an exception

Locks and objects

- There is only **one lock per object**
- If two threads call synchronized methods of the same object?
  - Only one can execute immediately
    - The other has to wait until the lock is released

The contents of this slide-set are based on the following references

- *Java Threads. Scott Oaks and Henry Wong. . 3rd Edition. O'Reilly Press. ISBN: 0-596-00782-5/978-0-596-00782-9.* [Chapters 3, 4]