

**Recruiter: could you give me an example of deadlock situation?**

**Programmer: Hire me and then I'll tell you**



# Lab 12: Monitors

## CS 2112 Fall 2024

December 2 / 4, 2024

# Threads

- ▶ Multiple threads may execute within the same program
- ▶ This is called called multithreading

# Race Conditions

```
1 class TugOfWar {  
2     int position = 0;  
3     void pullLeft() {  
4         position--;  
5     }  
6     void pullRight() {  
7         position++;  
8     }  
9 }
```

If thread 1 runs `pullLeft()` a thousand times and thread 2 runs `pullRight()` a thousand times, where does the rope end up?

# Locks

A lock, or a mutex (mutual exclusion object), is a mechanism that stops multiple threads from executing code at the same time.

Think of the lock as a physical object. Only one thread is allowed to “hold” the lock at any one point in time. Pass the lock object to the `synchronized` keyword to acquire it. Java automatically releases the lock at the end of the block.

```
1 public void run() {  
2     synchronized(mutex) {  
3         // Do unsafe mutable operation  
4     }  
5 }
```

## Using Mutexes

Note that the existence of a lock does not inherently protect your data. It's the job of the programmer to make sure that any unsafe calls acquire the mutex first.

A common pattern is to keep unsafe variables as private instance variables inside a class, and then enforce all accesses to those variables through publicly exposed methods in the class.

# Java Mutexes

Java allows any object to be used as a mutex.

Since the common use case involves keeping all lock acquisitions inside one object, it's customary to use `this` as the lock.

```
1 public void safeMethod() {  
2     synchronized(this) {  
3         // Do unsafe mutable operation  
4     }  
5 }
```

Since this is so common, using the `synchronized` keyword in the method header is syntactic sugar for the same.

```
1 public synchronized void safeMethod() {  
2     // Do unsafe mutable operation  
3 }
```

# Busy-Waiting

What happens if your multi-threaded code needs to wait on some condition being true?

One Solution:

```
1 public synchronized void waitingMethod() {  
2     while(!condition) { }  
3     // Do thing requiring condition  
4 }
```

Do you see any problems with this?



# Why Busy-Waiting Sucks

- ▶ Wastes CPU cycles
- ▶ Can cause deadlocks\* if you hold a mutex
- ▶ It's bad

\* Remember that a deadlock is when two threads both hold one lock and are both waiting on a lock held by each other

# Condition Variables

High Level Overview:

Mechanism for a thread to release its lock and wait on a condition.  
Automatically reacquire the lock when the condition becomes true.

Theoretically, you can have one condition variable for every  
condition you want to wait on.

# Condition Variables in Java

Java only allows one condition variable per object.

You can call the following methods from the Object API:

```
1 public void wait() // Makes this thread wait
2 public void notify() // Wake up one waiting thread
3 public void notifyAll() // Wake up all waiting threads
```

These methods can only be called by a thread that currently owns the object's lock.

# Using Monitors

A couple of best practices:

- ▶ You should wrap your `wait` commands inside a `while` loop, since you might want to wait on multiple conditions.

```
1 while(!condition1 || !condition2) {  
2     wait();  
3 }
```

- ▶ Call `notifyAll()` instead of `notify()` unless you have a very good reason to only wake up one thread, since there's no guarantee which thread is woken up.

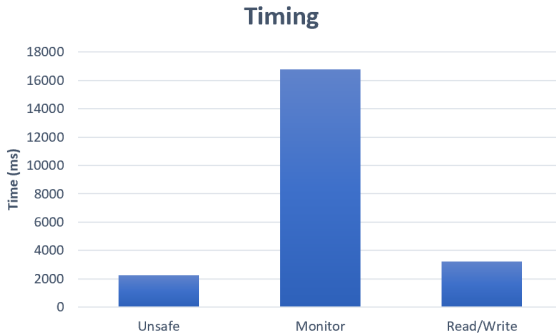
# Reentrant Locks

- ▶ Reentrant locks allow a single thread to acquire the same lock multiple times
- ▶ This allows one synchronized method to call another on the same object without getting stuck
- ▶ Each mutex keeps track of the number of times the thread has acquired the mutex and is only released once the holding thread releases it the same number of times.

# Read / Write Locks

- ▶ Any number of readers can hold the lock.
- ▶ Only one writer can hold it.
- ▶ Readers can starve out writers, so we need to stop new readers from joining when writers show up.

# Performance



## Exercise

Download the lab code and import it into IntelliJ (no Gradle :D)  
Run `MatrixTest.java` with the argument `unsafe`, `monitor`, or `rwlock` to run a bunch of threads without locks, with a normal lock, or with a read-write lock, respectively.  
Your task is to implement all the `TODOs` in `RWLock.java`.

Variables:

`numReaders` - Number of readers currently with the lock  
`numWritersWaiting` - Number of writers waiting for the lock  
`heldCount` - Number of times the writer has held this lock  
`writer` - The writer with the lock, or null



# The End + Thank You

It has been a pleasure working with all of you this semester. Best of luck on all your future endeavors, and thanks for coming to lab.

*Jonathan Gabor, Jonah Huang, Noah Schiff, Jake Silver, Michael Xing, James Zhang, Sean Zhang*