Threads and Locks

CSCI 5828: Foundations of Software Engineering
Lecture 09 — 09/22/2015

Goals

- Cover the material presented in Chapter 2, Day 1 of our concurrency textbook
  - Creating threads
  - Locks
  - Memory Models and Optimizations
    - The need for synchronization to get past the “memory barrier”
  - Deadlocks
    - Dining Philosophers
    - Alien Methods
    - How to Fix?

But First…

- I wanted to cover one additional Java threading construct that allows for synchronization between threads
  - We covered
    - the use of the keyword **volatile** to ensure that updates to a variable cross the memory barrier
    - the use of the keyword **synchronized** *on a method* to ensure that only one thread can execute the method at a time
- I forgot to cover another use of the **synchronized** keyword
  - where it can be applied *to an object* and then allow a section of code to be run while blocking other threads from accessing the object at that time
- This is referred to as a **synchronized block** since we synchronize on an object and then execute a block of code

Syntax

- To use the **synchronized** keyword in this fashion, you write code like this:
  - `synchronized (foo) {`
    - `a = foo.getValue();`
    - `if (a > b) {`
      - `foo.setSomeOtherValue(a -b);`
    - `}`
  - `}`
- This ensures that no other thread can access the object foo while this ***critical section*** is being executed as long as we assume that all modifications on foo are synchronized

Ensures?

- On the previous slide, I said
  - This **ensures** that no other thread can access the object foo while this ***critical section*** is being executed as long as we assume that all modifications on foo are synchronized
- But, does it really?
  - No. Because it requires ALL threads that access foo to wrap calls to foo in a synchronized block.
  - If **just one thread** that has a pointer to foo **forgets to wrap** calls to a critical section of code in a synchronized block then **all bets are off**
    - At that point, there can still be a **race condition** in your code because that rogue thread can make updates while other threads are in the critical section

Threads and Locks

Concurrency (the hard way)

- The most basic way to gain concurrency in your program is to use threads
  - Most programming languages provide access to a library that provides the ability to create threads
  - While the syntax will differ, at the conceptual level, you do the following

1. Identify code that needs to run in a separate thread
   - This code might exist in a run() method or be called by a method called run()
2. You create a thread that “wraps” the code in the previous step
3. You call start() on that new thread
4. At some point in the future, the wrapped code begins executing in a new thread of control

In Java

```
• public class HelloWorld {
   • public static void main(String[] args) throws InterruptedException {
      • Thread myThread = new Thread() {
         • public void run() {
            • System.out.println(“Hello from new thread”);
         • }
      • };
      • myThread.start();
      • Thread.yield()
      • System.out.println(“Hello from main thread”);
      • myThread.join();
   • }
• }
```

- To compile: javac -d build HelloWorld.java
- To run: java -classpath build HelloWorld

Using the book's source code (I)

- To run the book examples, it assumes you have Maven installed
  - <http://maven.apache.org/index.html>
- On Mac OS X, if you have Homebrew installed
  - <http://brew.sh>
- You can install Maven with this command
  - brew install maven
- You then need to set your Maven 3 environment variable
  - export M3_HOME=/usr/local/Cellar/maven/3.3.3
- Other platforms: download Maven 3 and set the M3_HOME to point at it
  - NOTE: Be sure to include the Maven bin directory in your path and make sure your JAVA_HOME variable is also set correctly

Using the book's source code (II)

- You can then go into any directory in the book's source code for Chapter 2 and type
  - mvn compile
  - mvn exec:java
- In the first step, Maven will download any packages it needs to compile the software and then compile it
  - In the second step, Maven will execute the program
- You will discover that Maven is "chatty". You can reduce the noise by including the -q flag (the quiet flag)
  - mvn -q compile

Using the book's source code (III)

- Note: you can obtain the book's source code by first going here
  - <https://www.pragprog.com/book/pb7con/seven-concurrency-models-in-seven-weeks>
- Then click on the small link near the top of the page that says "Source Code"
  - <https://www.pragprog.com/titles/pb7con/source_code>
- Then download either the .zip or .tar.gz file (the contents are the same), unpack the archive, and you're ready to go!

In Ruby

- Just to show that you can achieve the same effect in a different language

```
t = Thread.new { puts "Hello from new thread" }
Thread.pass
puts "Hello from main thread"
t.join
```

- Just put this code in a file and invoke via “ruby hello_world.rb”

Looking at Locks

- Our book starts with an example similar to what I showed in Lecture 8
  - The program has a class to store an integer value called Counter
  - It instantiates a single instance of that class
  - It creates a second class called CountingThread that extends Thread
    - In its run() routine, it loops 10 thousand times and increments Counter
- It creates two instances of CountingThread, starts them, and joins them
  - Calling join() on a thread means “Block me until this thread has stopped running”
- It then prints out the final result which is not 20,000 because of the race condition between the two threads

To “fix” this code, try synchronized

- The first change to this code is the obvious one
  - Change the Counter class so that its increment() routine is synchronized
    - `class Counter {`
      - `private int count = 0;`
      - `public synchronized void increment() { ++count; }`
      - `public int getCount() { return count; }`
    - `}`
- This makes sure that if two threads are calling increment(), they have to wait their turn
  - If we run the program now, the final count is 20K. Of course, at this point, we would be better off just writing a single threaded program!

Puzzle: Memory Models and Optimizations (I)

- The book mentions that the first attempt to fix the Counter program has a subtle bug and then introduces a program that can reveal the nondeterminism that governs concurrent programs
  - One thread does this
    - `answer = 42`
    - `answerReady = true;`
  - Another thread does this:
    - `if (answerReady)`
      - `System.out.println("The meaning of life is: " + answer);`
    - `else`
      - `System.out.println("I don't know the answer”);`
- The main program starts both threads and waits for them to finish

Puzzle: Memory Models and Optimizations (II)

- The book mentions that we can expect to see both outputs
  - On my computer, if I run Java in server mode, I see the first output
    - However, if I run Java in client mode, I do see the other output!!
- The book mentions that in this scenario
  - the compiler (javac), the run-time (java), and the hardware (the CPU)
    - all have the option to reorder the two assignment statements
- It also raises the issue of “memory barriers” and says that with the following code, Thread 2 may never stop running, as we saw in Lecture 8

```
public void run() {
  while (!answerReady) Thread.sleep(100);
  System.out.println("The meaning of life is: " + answer);
}
```

Puzzle: Memory Models and Optimizations (III)

- It says
  - *If your first reaction to this is that the compiler, JVM, and hardware should keep their sticky fingers out of your code, that's understandable. Unfortunately, it's also unachievable—much of the increased performance we've seen over the last few years* ***has come from exactly these optimizations****. Shared-memory parallel computers, in particular,* ***depend on them****. So we're stuck with having to deal with the consequences.*
- Memory visibility
  - In order for one thread to see changes in memory made by another thread
    - **both** threads (the reading thread and the writing thread) need to use synchronization of some kind
      - It's not enough to have just one of the threads use synchronization
  - Thus in the counting program, getCount() needs to be synchronized

Multiple Locks and Deadlock (I)

- We've now seen race conditions and memory visibility as two types of problems in concurrent programs
    - Let's take a look at Deadlock with the Dining Philosophers example
- Making the two methods of the Counter class synchronized is an example of placing two locks in the code, one for reading and one for writing
- In actuality, we have a single lock (called an *intrinsic lock*) on the object but it is accessed in two methods, one that reads a value and the other that updates the value
- An intrinsic lock is a lock that the JVM provides for each object automatically; as we will see, it is must better to use explicit locks
    - Although it is WAY better to get away from threads and locks altogether!

Multiple Locks and Deadlock (II)

- You might think that one solution to concurrency problems is to *make* ***every method*** *on a* ***shared object synchronized***
- BUT
    - such programs are terribly inefficient
        - blocking threads when they don't need too;
- plus, it can lead to deadlock (as we will see next);
    - and you can STILL have race conditions
        - as we will see on an upcoming homework

Multiple Locks and Deadlock (III)

- Dining Philosophers
    - Multiple threads with shared resources
    - Five philosophers (threads) sit at a table
        - There are five chopsticks (resources) on the table
        - If a philosopher wants to eat, they acquire the chopstick to their left and their right and eat
            - otherwise they sit at the table thinking quietly
- This situation can deadlock if all philosophers decide to eat at the same time and they each pick up the chopstick to their left
    - They now all want to pick up the chopstick on their right; they're stuck!

Implementation

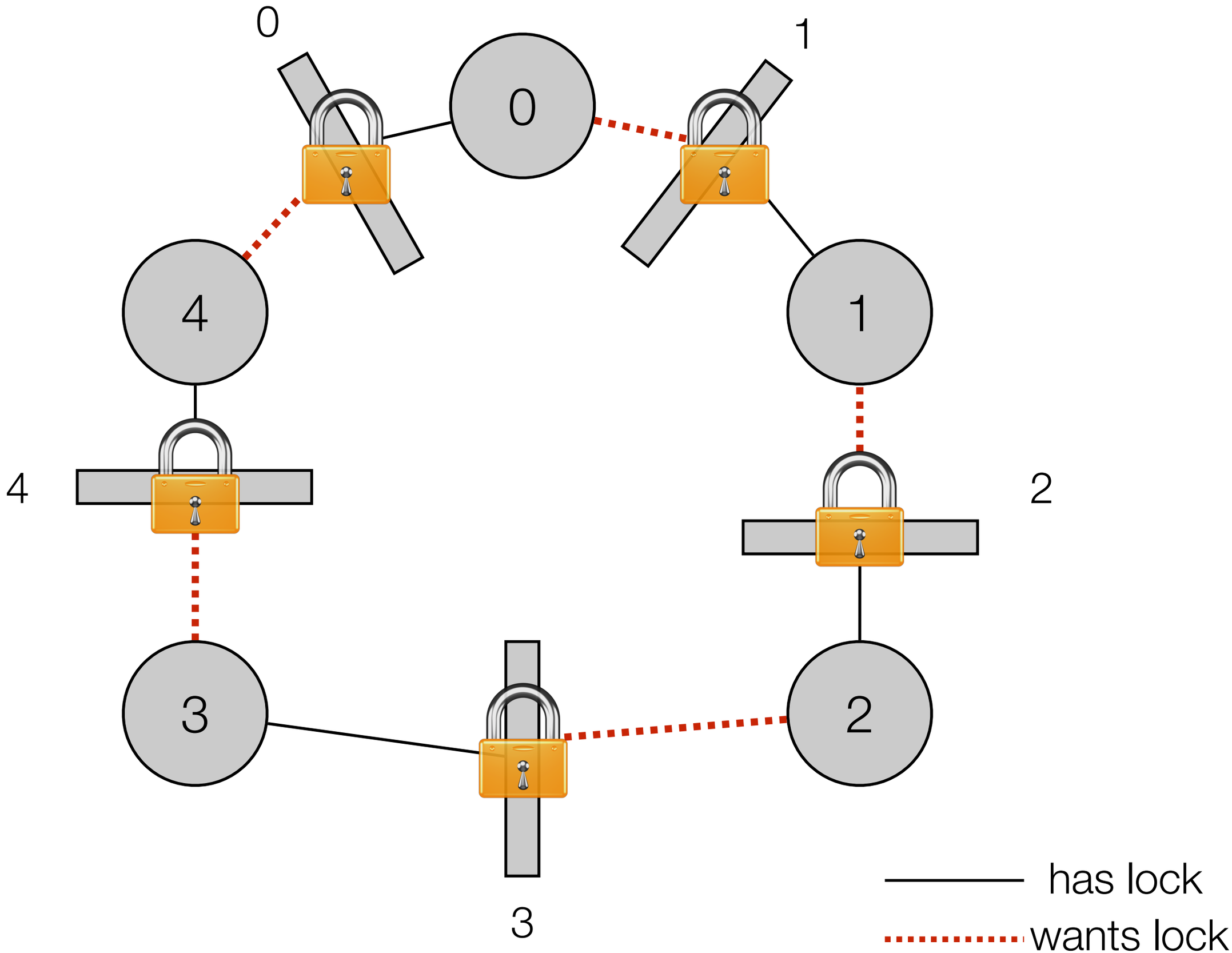
- Chopstick is implemented as a class that keeps track of an id
  - It receives its id via its constructor and can return its id via getId()
- Philosopher is implemented as a class that is a subclass of Thread
  - It has instance variables for its right and left chopsticks which it receives via its constructor
  - It makes use of the synchronized (object) syntax that we saw at the start of this lecture, to acquire each chopstick
  - It sits in a loop where it thinks for about a second, gets its chopsticks, and eats for about a second
- DiningPhilosophers is a class that contains the main() routine
  - It creates five chopsticks and five philosophers, gives each philosopher the correct chopsticks, starts the philosophers, and waits for them to finish
    - They never will… we're just letting them run until deadlock occurs

Contention

- When you first run the Dining Philosophers program, it may happily run for a long time
  - The default implementation is to have the philosophers pick a random number from 1 and 1000 and sleep that many milliseconds for “sleeping” and “eating”
    - That’s a relatively long time for the scheduler and so there’s **low contention** on the locks associated with each chopstick
      - As a result, there’s not a lot of opportunity for deadlock to occur
- If you want to see the program lock up right away, simply decrease the amount of time the philosophers spend sleeping and eating
  - I changed my copy of the program to pick a number between 1 and 10 milliseconds
    - We now have **high contention** on the locks and the program quickly deadlocks

How to Fix? (I)

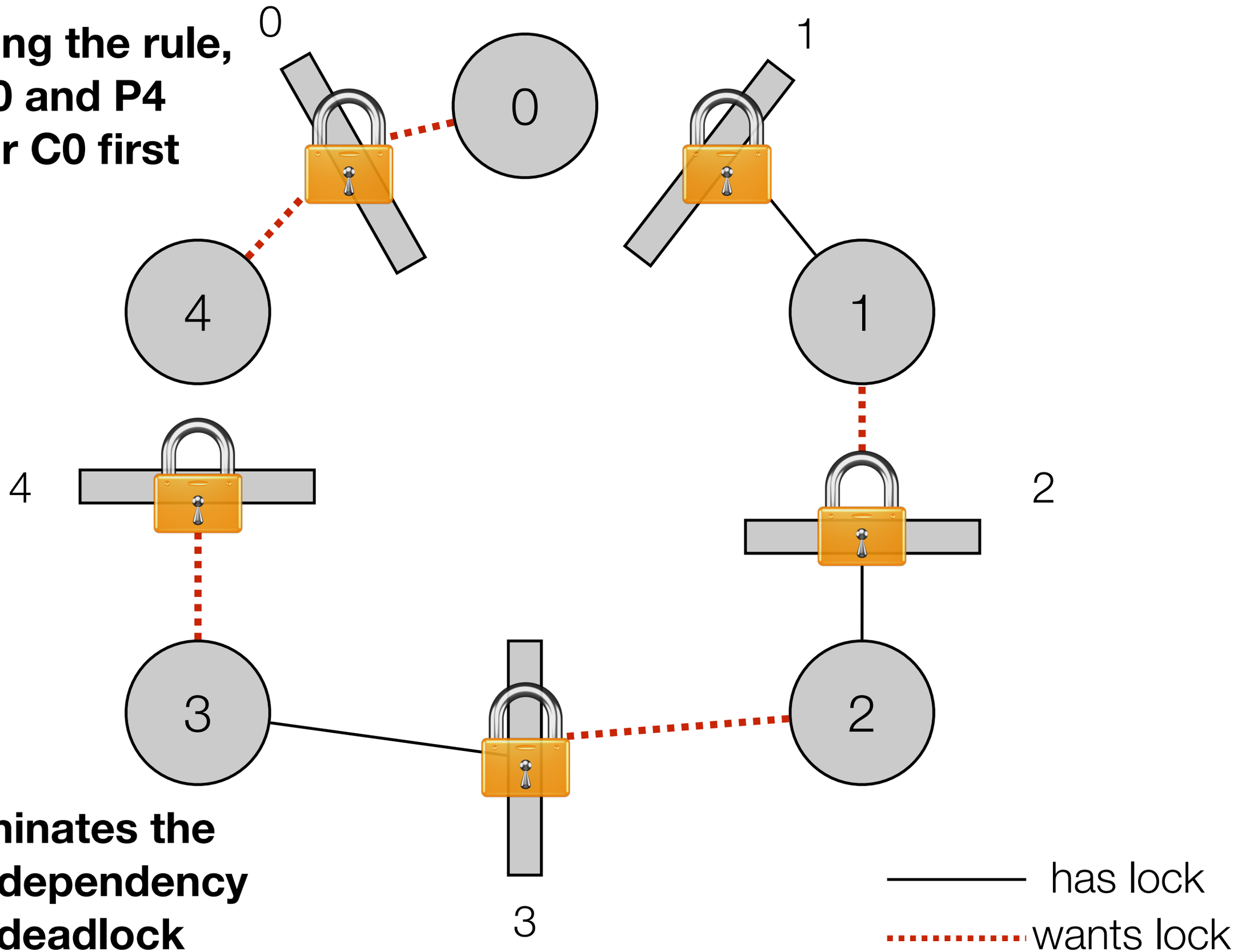
- Fortunately, most deadlock problems can be solved with this rule
  - Always acquire resources in the same order
- In the first version of the program, it was possible for this situation to occur
  - Philosopher 0 has Chopstick 0 and wants Chopstick 1
  - Philosopher 1 has Chopstick 1 and wants Chopstick 2
  - Philosopher 2 has Chopstick 2 and wants Chopstick 3
  - Philosopher 3 has Chopstick 3 and wants Chopstick 4
  - Philosopher 4 has Chopstick 4 and wants Chopstick 0
    - Deadlock!



How to Fix? (II)

- In the second version of the program, we follow the rule and…
  - P0 has been assigned Chopsticks 0 and 1
  - P4 has been assigned Chopsticks 4 and 0
  - P0 and P4 both decide they are hungry
    - They **BOTH** reach for Chopstick 0 first
      - One of them gets it
      - One of them blocks
    - If P0 gets it, then P4 is blocked **BUT** Chopstick 4 is still free!
      - This lets P3 eat and finish which lets P2 eat and finish…
        - which eventually lets P1 each and finish unblocking P4

Following the rule, both P0 and P4 grab for C0 first



This eliminates the circular dependency and the deadlock

has lock

wants lock

Alien Methods

- Our book has a section called “The Perils of Alien Methods”
  - If you find it confusing, you’re not alone!
  - It is not well explained AND, even worse, the example program does NOT demonstrate deadlock in this situation
- The basic idea is the following
  - You write code that has a resource that is potentially shared by one or more objects
    - You engineer your code to make sure that they follow the proper rules for synchronizing access to the shared resource
    - BUT, your code also provides an extension mechanism, that allows code NOT written by you to be “plugged into” your system
  - If you invoke that code and it accesses your shared resource from another thread, deadlock can occur

The Example (I)

- The “alien methods” example has the following set-up
  - A “downloader” object is given a URL to a large data file on the web
    - The “downloader” object implements a “listener” mechanism that allows other code to “plug in” and get notifications about the progress of the download
      - The listener methods are all marked as synchronized to allow multiple threads to add/remove listeners during the download
- The example code implements a simple plug-in that simply prints out the progress in bytes for each notification that it receives
  - The example code does NOT show how this could lead to deadlock
- Also not mentioned: it’s trying to download a 12GB file!

The Example (II)

- So, I created a modified version of the program to demonstrate how deadlock could occur
  - My modification does the following
    - The listener creates an object called AlienThread which is a subclass of Thread. It passes to AlienThread a reference to the downloader and itself
    - The listener then calls start() on that object and then join()
      - As a result, this listener's event handler becomes "long running", holding on to the lock that was acquired by the downloader's synchronized listeners methods
    - Then in the run() method of the thread, we call downloader's removeListener() method. Since this method is synchronized, we try to acquire the lock and DEADLOCK results

Source Code for AlienThread

```
• public class AlienThread extends Thread {
    • private Downloader downloader;
    • private ProgressListener l;
    • AlienThread(Downloader d, ProgressListener l) {
        • this.downloader = d;
        • this.l = l;
    • }
    • public void run() {
        • downloader.removeListener(l);
    • }
• }
```

Modified Code for HttpDownload

```
• final Downloader downloader = new Downloader(from, “download.out");

• final ProgressListener l = new ProgressListener() {
   • public void onProgress(int n) {
      • try {
         • System.out.print(“\r"+n);
         • System.out.flush();
         • AlienThread alien = new AlienThread(downloader, this);
         • alien.start(); alien.join();
      • } catch (InterruptedException ex) {
      • }
   • }
   • public void onComplete(boolean success) {}
• };
• downloader.start();
• downloader.addListener(l);
• downloader.join();
```

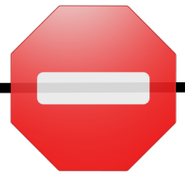
The Example (III)

Thread 1

```
private synchronized void updateProgress(int n) {
  for (ProgressListener listener: listeners)
    listener.onProgress(n);
}
```

Thread 2

```
public synchronized void removeListener(ProgressListener listener) {
  listeners.remove(listener);
}
```



DEADLOCK!

<<create>> start() join()

Downloader

ProgressListener

Note: creating a new thread was required! (I)

- To make a deadlock happen, I **had** to create a new thread before calling the Downloader's removeListener() method
- That's because the lock that Java creates for each object is a ReentrantLock
  - When you call a synchronized method on an object, Java asks for that object's lock and tries to lock it
  - If that synchronized method turns around and calls another synchronized method on the same object, it has to acquire the lock again
    - Deadlock would occur right there even with a single thread of control if the lock wasn't reentrant.

Note: creating a new thread was required! (II)

- Instead, the reentrant lock
  - detects that the same thread is trying to lock it again
  - it allows that to happen but it keeps track of the number of times that thread calls lock()
    - it then requires the same number of unlock() calls by the same thread before it considers itself “unlocked”
- In our example, if we did not create a second thread of control, then when the listener called removeListener on the Downloader, the reentrant lock would have allowed the call to happen
  - And, instead, our program would die with a concurrent modification exception; that is, we would be trying to remove a listener from the list while updateProgress() was iterating over that list

How to fix? (I)

- To fix this type of deadlock, you need to make sure that you do not call the “alien method” while holding a lock on a potentially shared resource
    - With our example, this means that we remove the synchronized keyword on the updateProgress method. That way, we are not holding the lock on the Downloader (our shared resource) when we call the listeners
        - Instead, INSIDE of the method, we temporarily acquire the lock and use that lock to copy the current list of listeners. We then release the lock and invoke the listeners onProgress() methods via the copy of the list
        - This is called making a defensive copy of the list
            - It also reduces the amount of time we hold the lock and therefore we reduce contention
- We can verify that this solution works with my variation of the example

How to fix? (II)

- If we apply my changes to the HttpDownloadFixed directory
  - We will see our progress indicator publish one event
    - By printing 1024 to standard out
    - It then removes itself from the set of listeners
      - so it no longer receives any updates
- But, by looking at the file system, we can see that file is still downloading
  - The system did not DEADLOCK
  - The solution worked!

Summary

- In this lecture, we've encountered an introduction to doing concurrency the hard way via threads and locks
- We've discussed the various problems you might encounter
    - race conditions
    - memory visibility (as well as operation re-ordering)
    - deadlocks
- To solve the first two problems
    - you need locks and synchronization
- To solve the third
    - you need to either acquire shared resources in the same order across all threads or you need to do what you can to reduce lock contention

Coming Up Next

- Lecture 10: Concurrency: Threads and Locks, Part Two
    - Material from Day 2 and Day 3