

CMSC 132: Object-Oriented Programming II



Synchronization in Java

Department of Computer Science
University of Maryland, College Park

Multithreading Overview

■ Motivation & background

■ Threads

■ Creating Java threads

■ Thread states

■ Scheduling

■ Synchronization

■ Data races ←

■ Locks

■ Deadlock



Data Race

■ Definition

- Concurrent accesses to same shared variable, where at least one access is a write

■ Properties

- Order of accesses may change result of program
- May cause intermittent errors, very hard to debug

■ Example

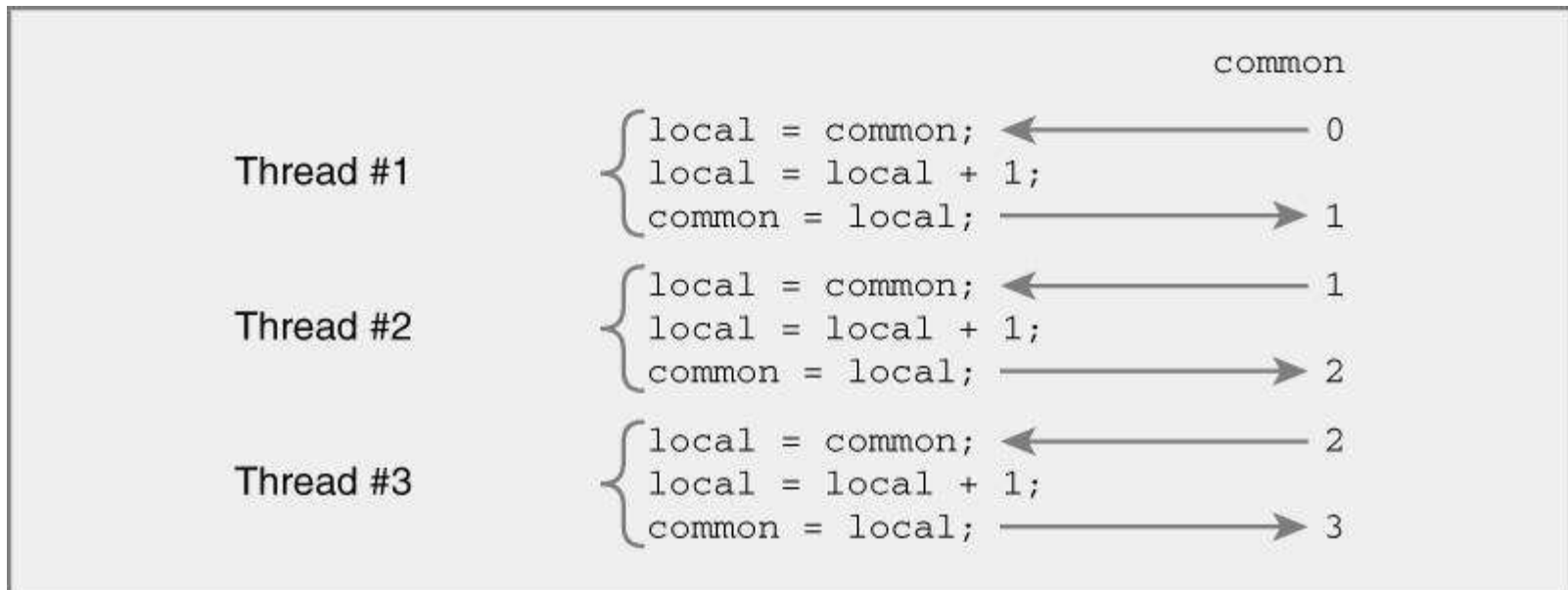
```
public class DataRace extends Thread {  
    static int x;    // shared variable x causing data race  
    public void run() { x = x + 1; } // access to x  
}
```

Data Race Example

```
public class DataRace extends Thread {
    static int common = 0;
    public void run() {
        int local = common; // data race
        local = local + 1;
        common = local; // data race
    }
    public static void main(String[] args) throws InterruptedException {
        int max = 3;
        DataRace[] allThreads = new DataRace[max];
        for (int i = 0; i < allThreads.length; i++)
            allThreads[i] = new DataRace();
        for (DataRace t : allThreads)
            t.start();
        for (DataRace t : allThreads)
            t.join();
        System.out.println(common); // may not be 3
    }
}
```

Data Race Example

■ Sequential execution output



Data Race Example

■ Concurrent execution output (possible case)

The diagram illustrates a data race scenario with three threads and a shared variable 'common'. The threads perform the following operations in sequence:

- Thread #1: `local = common;` (reads 0)
- Thread #2: `local = common;` (reads 0)
- Thread #3: `local = common;` (reads 0)
- Thread #1: `local = local + 1;`
- Thread #2: `local = local + 1;`
- Thread #3: `local = local + 1;`
- Thread #1: `common = local;` (writes 1)
- Thread #2: `common = local;` (writes 1)
- Thread #3: `common = local;` (writes 1)

The final value of 'common' is 1.

■ Result depends on thread execution order!

Synchronization

■ Definition

- Coordination of events with respect to time

■ Properties

- May be needed in multithreaded programs to eliminate **data races**
- Incurs runtime overhead
- Excessive use can reduce performance

Lock

■ Definition

- Entity can be held by only one thread at a time

■ Properties

- A type of synchronization
- Used to enforce **mutual exclusion**
 - Thread can acquire / release locks
 - Only 1 thread can acquire lock at a time
- Thread will wait to acquire lock (stop execution)
 - If lock held by another thread
- Used to implement **monitors**
 - Only 1 thread can execute (locked) code at a time



Synchronized Objects in Java

■ Java objects provide locks

■ Apply **synchronized** keyword to object

■ Will acquire / release lock associated with object

■ Mutual exclusion for code in synchronization **block**

■ Example

```
Object x = new Object();
```

```
block { synchronized(x) { // acquire lock on x on entry
    ...                // hold lock on x in block
}                      // release lock on x on exit
```

Synchronized Methods In Java

- Apply synchronized keyword to method
 - Mutual exclusion for entire body of method
 - Synchronizes on object invoking method

- Example

block



```
synchronized foo() { ...code... }
```

// shorthand notation for

```
foo() {  
    ↓  
    synchronized (this) { ...code... }  
}
```

Synchronized Methods In Java

```
public synchronized void enqueue( Object item ) {  
    // Body of method goes here  
}
```



Shorthand notation for

```
public void enqueue( Object item ) {  
    synchronized ( this ) {  
        // Body of method goes here  
    }  
}
```

Locks in Java

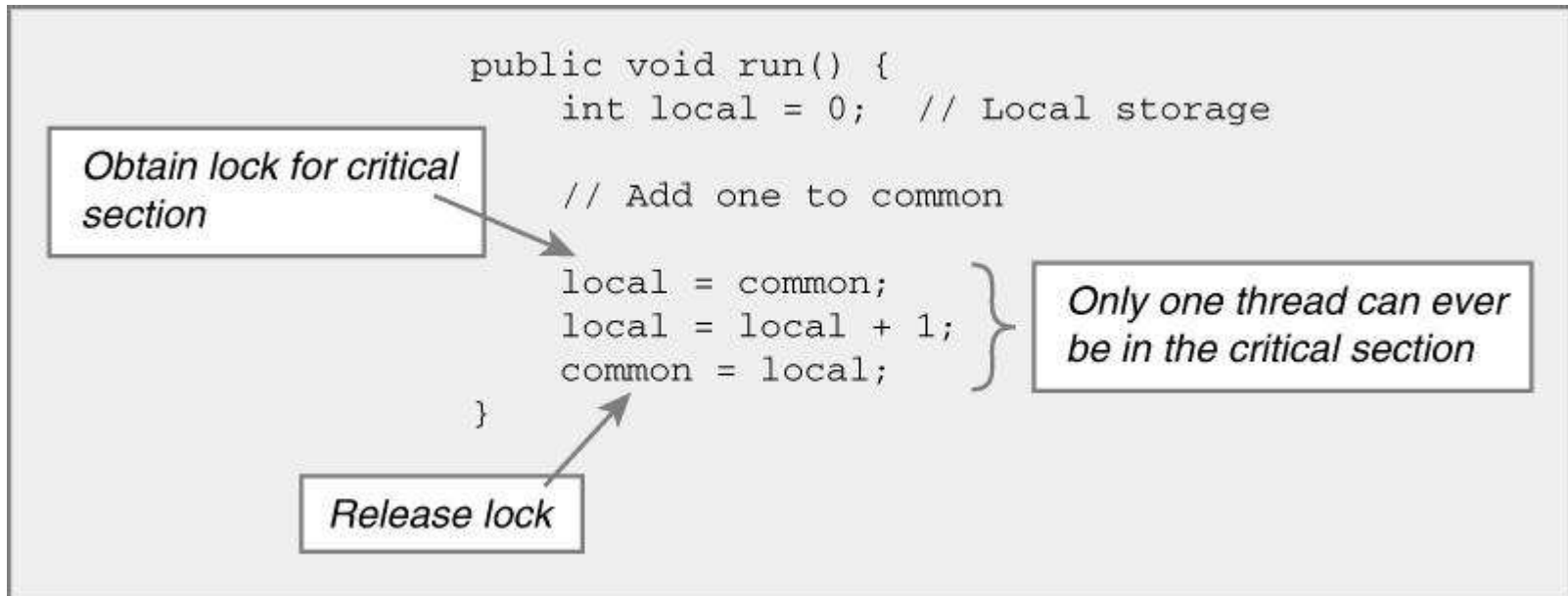
■ Properties

- No other thread can get lock on x while in block
- Other threads can still access/modify x!
- Locked block of code $\Rightarrow$ **critical section**

■ Lock is released when block terminates

- End of block reached
- Exit block due to return, continue, break
- Exception thrown

Synchronization Example



Lock Example

```
public class DataRace extends Thread {  
    static int common = 0;  
    static Object o;           // all threads use o's lock  
    public void run() {  
        synchronized(o) {    // single thread at once  
            int local = common; // data race eliminated  
            local = local + 1;  
            common = local;  
        }  
    }  
    public static void main(String[] args) {  
        o = new Object();  
        ...  
    }  
}
```

Synchronization Issues

- 1. Use same lock to provide mutual exclusion**
- 2. Ensure atomic transactions**
- 3. Avoiding deadlock**

Issue 1) Using Same Lock

■ Potential problem

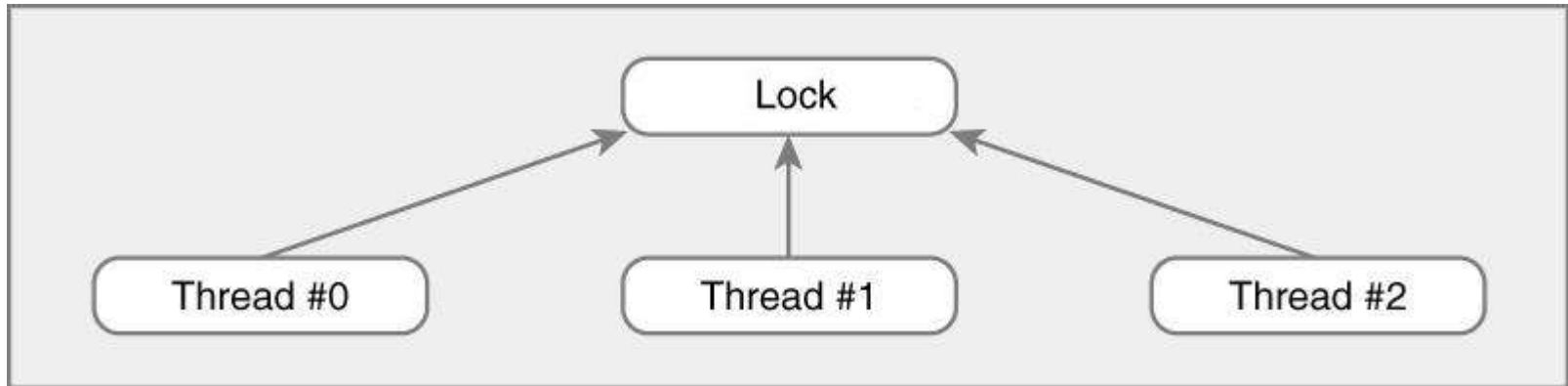
- Mutual exclusion depends on threads acquiring same lock
- No synchronization if threads have different locks

■ Example

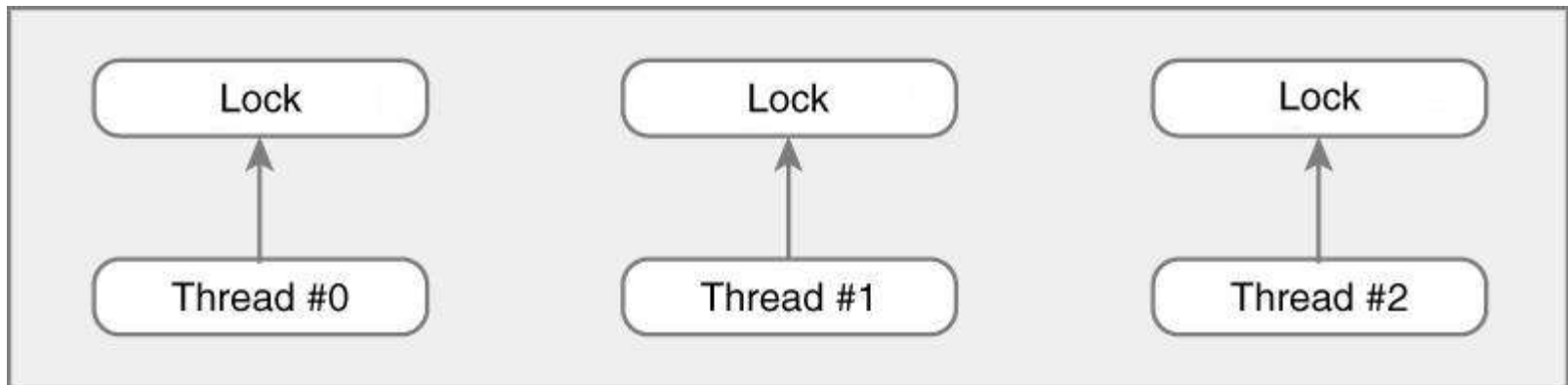
```
foo() {  
    Object o = new Object(); // different o per thread  
    synchronized(o) {  
        ... // potential data race  
    }  
}
```

Locks in Java

- **Single lock for all threads (mutual exclusion)**



- **Separate locks for each thread (no synchronization)**



Lock Example – Incorrect Version

```
public class DataRace extends Thread {  
    static int common = 0;  
    public void run() {  
        Object o = new Object(); // different o per thread  
        synchronized(o) {  
            int local = common;    // data race  
            local = local + 1;  
            common = local;        // data race  
        }  
    }  
    public static void main(String[] args) {  
        ...  
    }  
}
```

Issue 2) Atomic Transactions

■ Potential problem

- Sequence of actions must be performed as single **atomic transaction** to avoid data race
- Ensure lock is held for duration of transaction

■ Example

```
synchronized(o) {  
    int local = common;  
    local = local + 1;  
    common = local;  
}
```

} // all 3 statements must
// be executed together
// by single thread

Lock Example – Incorrect Version

```
public class DataRace extends Thread {  
    static int common = 0;  
    static Object o;           // all threads use o's lock  
    public void run() {  
        int local;  
        synchronized(o) {  
            local = common;  
        }  
        synchronized(o) {  
            local = local + 1;  
            common = local;  
        }  
    }  
}
```

} // transaction not atomic
// data race may occur
// even using locks

Issue 3) Avoiding Deadlock

■ Potential problem

- Threads holding lock may be unable to obtain lock held by other thread, and vice versa
- Thread holding lock may be waiting for action performed by other thread waiting for lock
- Program is unable to continue execution (**deadlock**)

Deadlock Example 1

Object **a**;

Object **b**;

```
Thread1() {  
    synchronized(a) {  
        synchronized(b) {  
            ...  
        }  
    }  
}
```

```
Thread2() {  
    synchronized(b) {  
        synchronized(a) {  
            ...  
        }  
    }  
}
```

// Thread1 holds lock for a, waits for b

// Thread2 holds lock for b, waits for a

Deadlock Example 2

```
void swap(Object a, Object b) {  
    Object local;  
    synchronized(a) {  
        synchronized(b) {  
            local = a; a = b; b = local;  
        }  
    }  
}
```

```
Thread1() { swap(a, b); } // holds lock for a, waits for b  
Thread2() { swap(b, a); } // holds lock for b, waits for a
```

Deadlock

■ Avoiding deadlock

- In general, avoid holding lock for a long time
- Especially avoid trying to hold two locks
 - May wait a long time trying to get 2nd lock

Thread-safe

- **Thread-safe** – Code is considered thread-safe if it works correctly when executed by multiple threads simultaneously.
- **Example: ArrayList is not thread-safe**

From Java API: “Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally.”

Synchronization Summary

- Needed in multithreaded programs
- Can prevents data races
- Java objects support synchronization
- Many other tricky issues
 - To be discussed in future courses