CMSC 132:
OBJECT-ORIENTED PROGRAMMING II

Synchronization in Java

Department of Computer Science
University of Maryland, College Park

Data Race: Definition and Properties

- **What is a Data Race?**
  A *data race* happens when:
  - Two or more threads access the same shared variable or resource (located on the heap),
  - **At least one access is a write**, and
  - The accesses are **not properly synchronized**.
- **Thread Memory Model**
  - Each thread has its **own stack** (local variables, method calls).
  - But threads **share the heap**, where objects like Maps, Sets, and Arrays are stored.
  - These shared objects are **mutable** — if one thread modifies them while another reads or writes, a data race can occur.
- **Why is it a Problem?**
  - The **execution order of threads is unpredictable**.
  - This may lead to different results on each run.
  - These bugs are **intermittent**, **hard to reproduce**, and **difficult to debug**.

Data Race Example

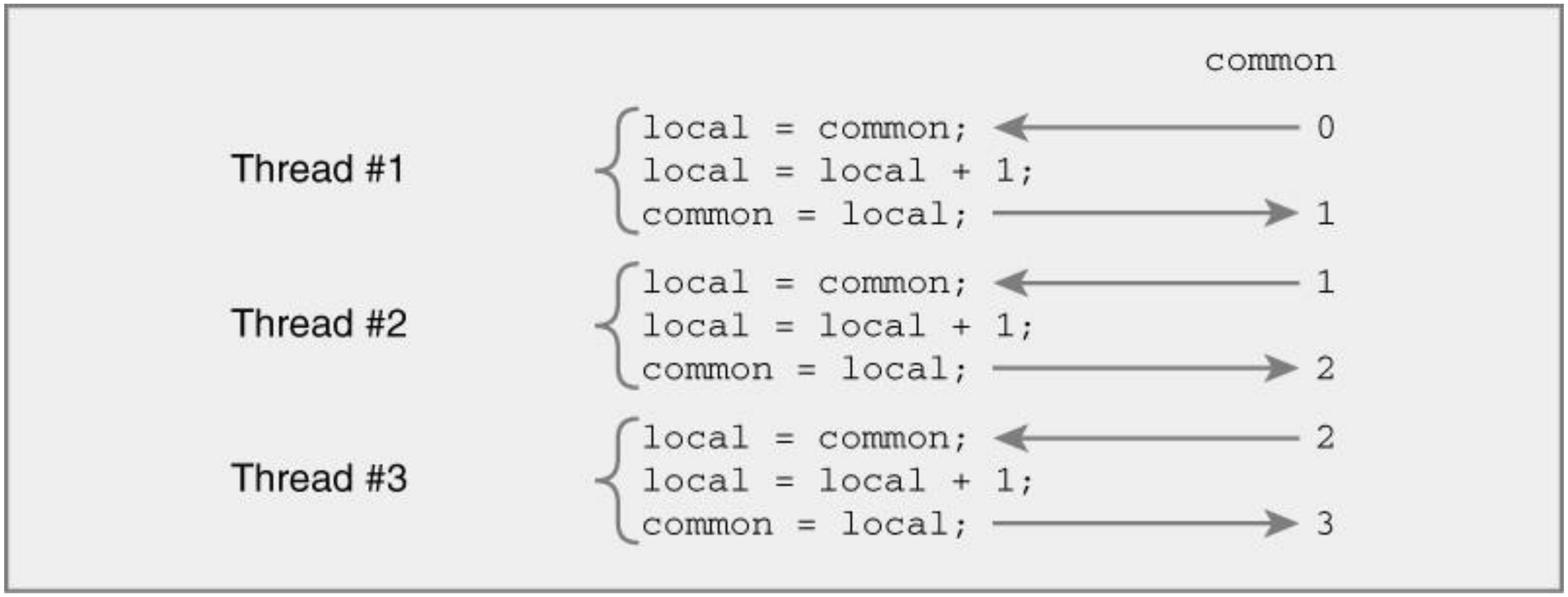
```
public class DataRace extends Thread {
    static int common = 0;
    public void run() {
        int local = common;   // data race: read
        local = local + 1;
        common = local;       // data race: write
    }
    public static void main(String[] args) throws InterruptedException {
 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

Notes about Data Race Example

- This program has a **data race** on the common variable:
  - Multiple threads read and write common without synchronization.
- The final value of common **may not be 3**, depending on the timing of thread execution.
- However, because **each thread does very little work**, the race condition might **not appear consistently** — the output **may appear correct by chance**.
- In practice, race conditions often **hide during testing** and only show up under heavy load or in production environments.
- **Warning: common++ does NOT fix the problem**
  Even though common++ looks like a single operation, it's actually **three low-level steps**:
  1. Read common from memory
  2. Increment the value
  3. Write the result back to memory
- Without synchronization, a thread can be **interrupted between any of these steps**, allowing another thread to read or write the same variable.
- This leads to **lost updates** and inconsistent results — classic symptoms of a **data race**.

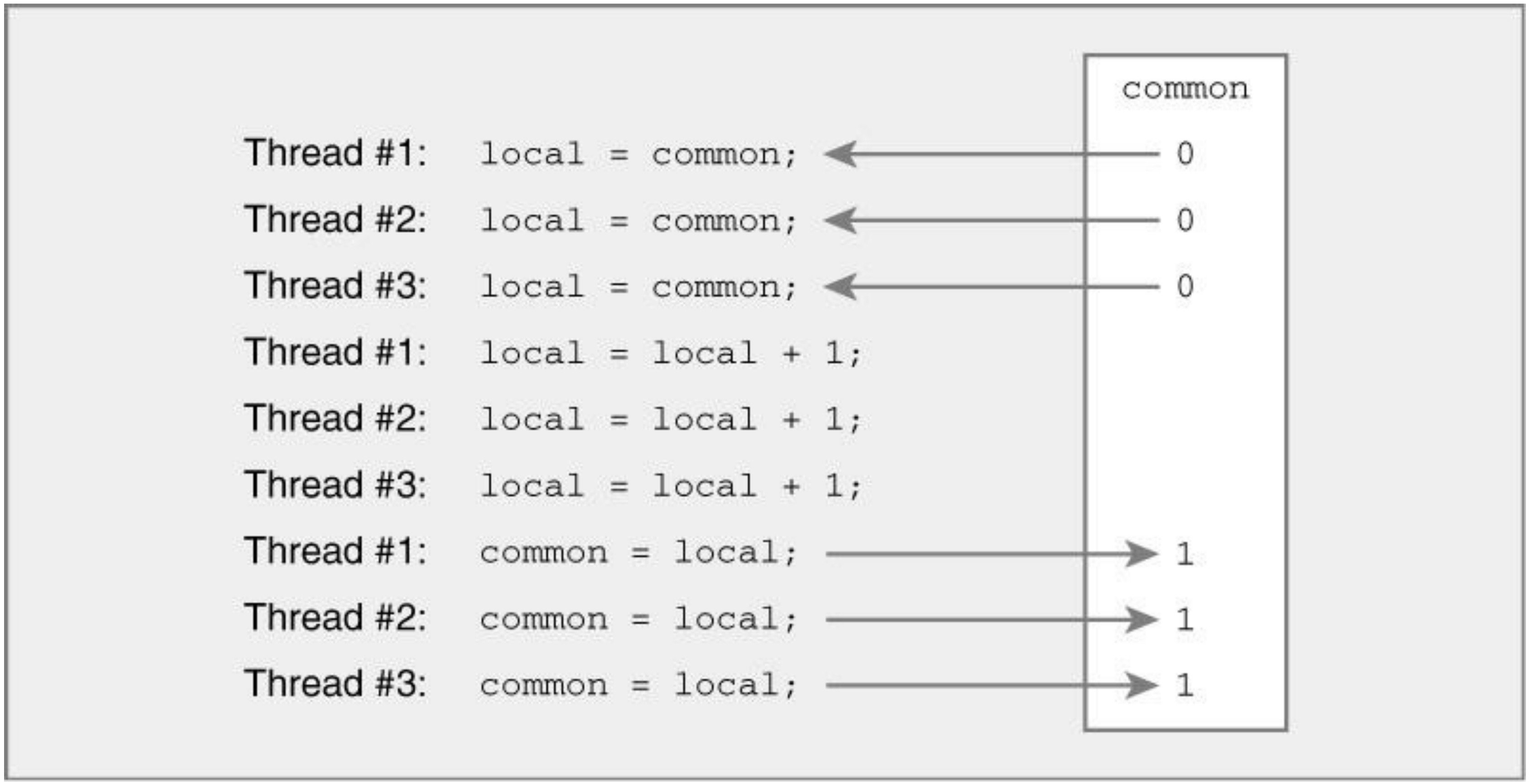
Data Race Example

- Sequential execution output



Data Race Example

- Concurrent execution output (possible case)


common
Thread #1: local = common; 0
Thread #2: local = common; 0
Thread #3: local = common; 0
Thread #1: local = local + 1;
Thread #2: local = local + 1;
Thread #3: local = local + 1;
Thread #1: common = local; 1
Thread #2: common = local; 1
Thread #3: common = local; 1

See `DataRaceExample in incCommon package`

Synchronization

- **Definition:**
  Synchronization in Java ensures the coordinated execution of threads, particularly in concurrent environments, to prevent issues like data races and inconsistent states.
- **Key Properties:**
  - **Prevents Data Races:** Synchronization is crucial to avoid multiple threads accessing shared data simultaneously, which could lead to inconsistent results.
  - **Runtime Overhead:** Using synchronization can incur performance penalties due to context switching and the need to acquire/release locks.
  - **Performance Trade-off:** Excessive synchronization can hinder performance, as it may cause threads to block unnecessarily, reducing throughput.

Locks in Java

A **lock** in Java is an object that ensures exclusive access to a critical section, allowing only one thread to access it at a time.

- **Key Properties:**
- **Type of Synchronization:** A lock is **a specific synchronization mechanism** used to prevent race conditions by ensuring mutual exclusion.
- **Mutual Exclusion:** Locks enforce mutual exclusion by allowing only one thread to enter the critical section at a time.
- **Critical Section:** The critical section is the code that accesses shared resources, such as updating a shared variable.
- **Thread Behavior:**
    - A thread can **acquire** or **release** a lock.
    - Only one thread can hold the lock at any given time. If the lock is already held by another thread, the requesting thread will block (pause execution) until the lock is released.

Note:

The term "critical section" here refers to the section of code that needs synchronization and not the "critical section" used in algorithmic complexity analysis.

Key Points:

- **Intrinsic Lock = Monitor Lock**: The lock provided by every Java object to ensure that only one thread can access the synchronized code at a time.
- **synchronized Keyword**: This keyword is used to acquire the intrinsic lock of the object specified. It can be applied to both methods and code blocks to provide mutual exclusion.

Using the synchronized Keyword in Java

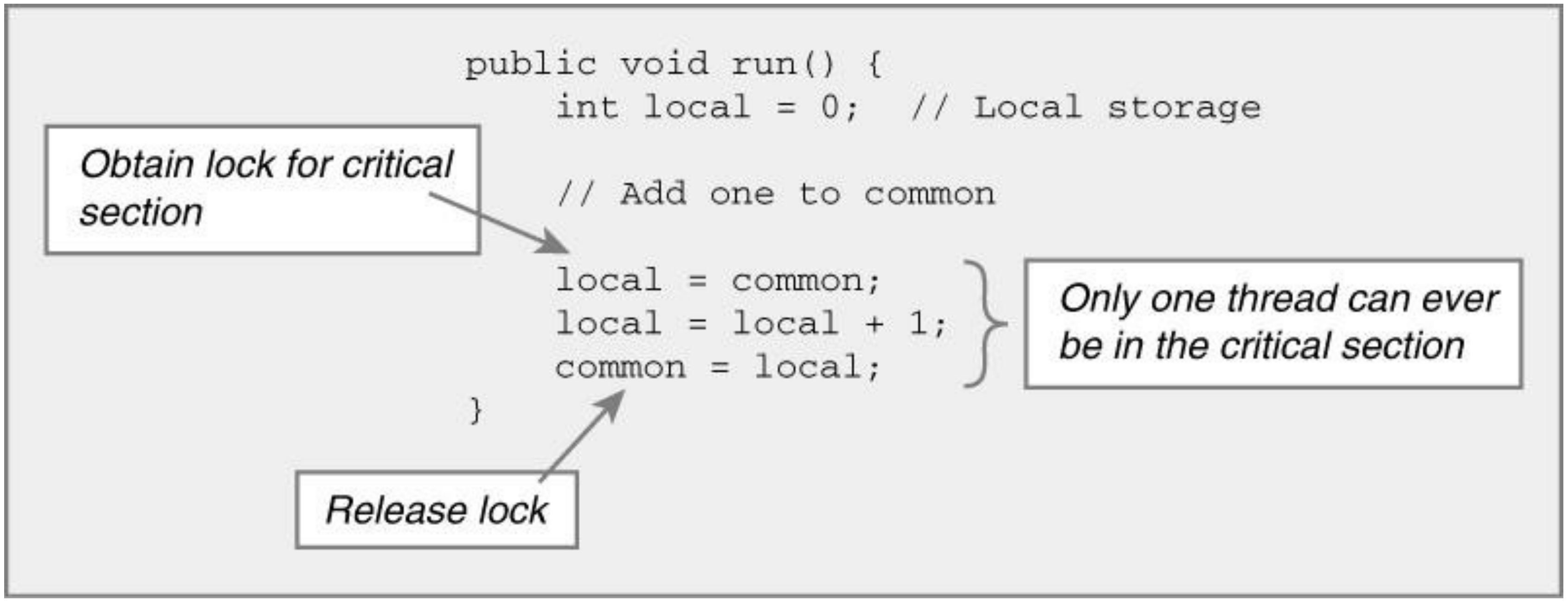
- **Key Concepts:**
- **Java Objects Have Locks:** Every Java object has an associated lock, which can be used for synchronization to ensure that only one thread can execute a critical section of code at a time.
- **Acquiring a Lock:**
  A thread acquires the lock of an object using the synchronized keyword. The syntax is as follows:

```
Object x = new Object();  // Any object can be used as the "locking object"
synchronized(x) {          // Attempt to acquire the lock on object x
    // Critical section: thread holds the lock on x during this block
}                          // Lock is released when the block exits
```

Lock Behavior:

- **Acquiring the Lock:**
  A thread can acquire the lock if no other thread currently holds it.
- **Blocking on the Lock:**
  If another thread holds the lock, the requesting thread will block and wait until the lock becomes available.
- **Releasing the Lock:**
  The lock is released as soon as the thread exits the synchronized block, regardless of how it exits (e.g., via return, exception, etc.).

Fixing Data Race In Our Example



```
public void run() {
    int local = 0;   // Local storage

    // Add one to common

    local  = common;
    local  = local + 1;
    common = local;
}
```

Obtain lock for critical section
Only one thread can ever be in the critical section
Release lock

See `NoDataRace` in incCommon package

Another Example (Account)

- We have a bank account shared by two kinds of buyers (Excessive and Normal)
- We can perform deposits, withdrawals and balance requests for an account
- Critical section → account access
- No Synchronization (Example: **noLock**)
  - Bad idea
- First solution (Example: **explicitLockObj**)
  - We use lockObj to protect access to the Account object
- Second solution (Example: **accountAsLockObj**)
  - Notice we don't need to define an object to protect the Account object as Account already has a lock
- Third Solution (Example: **lockObjInAccount**)
  - We are using lockObj to protect access to the Account object. The lock is now in the account class.
- Fourth Solution (Example: **usingThisInAccount**)
  - Notice we don't need to define an object to protect the Account object as Account already has a lock

Synchronized Methods In Java

If the **entire body** of a method is synchronized using the **current object's lock** (i.e., synchronized(this)), Java allows a shorthand using the synchronized keyword in the method declaration.

Syntax Comparison

- These two are equivalent:

```
// Shorthand: synchronized method
public synchronized void foo() {
    // method body is protected by this object's intrinsic lock
    ...
}
// Equivalent expanded version
public void foo() {
    synchronized (this) {
        // method body is protected by this object's intrinsic lock
        ...
    }
}
```

Notes

- Applies only to **instance methods**; it uses the **intrinsic lock** of the current object (this).
- Ensures **mutual exclusion** for the **entire method body** — only one thread can execute foo() on the same object at a time.
- For **static methods**, synchronized locks on the **Class object**, not this.
- Fifth Solution (Example: **synchronizedMethods)**