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

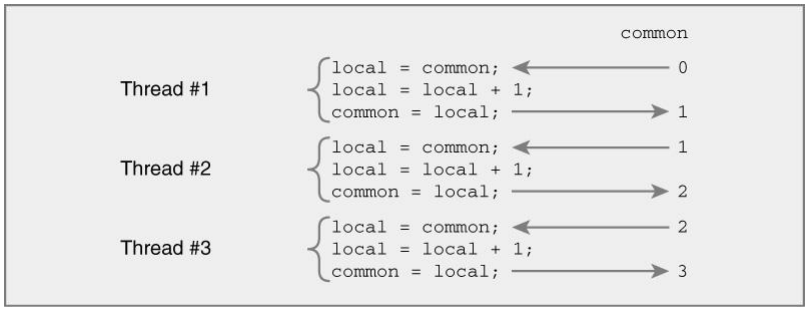
- Definition
  - Concurrent accesses to same shared variable/resource, where **at least one** access is a write
    - Resource → map, set, array, etc.
- Properties
  - Order of accesses may change result of program
  - May cause intermittent errors, very hard to debug

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
   }
}
```

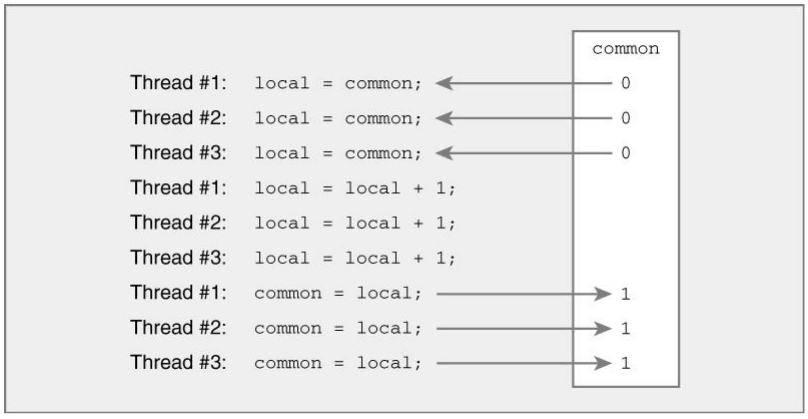
Data Race Example

- Sequential execution output



Data Race Example

- Concurrent execution output (possible case)



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 that can be held by only one thread at a time
- Properties
  - A type of synchronization
  - Used to enforce mutual exclusion so we can protect the critical section
    - Critical section in previous example was increasing common
    - Note: critical section should not be confused with the term critical section use for algorithmic complexity analysis
  - 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

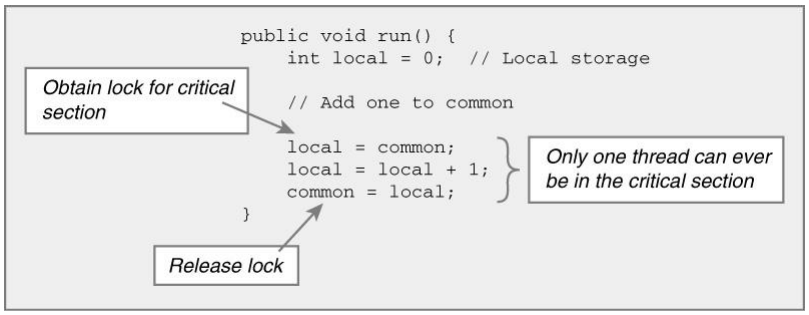
Synchronized Objects in Java

- Every Java object has a lock
- A lock can be held by only one thread at a time
- A thread acquires the lock by using synchronized
- Acquiring lock example

```
Object x = new Object();  // We can use any object as "locking object"
synchronized(x) {     // try to acquire lock on x on entry
        ...           // hold lock on x in block
}                     // release lock on x on exit
```

- When synchronized is executed
  - Thread will be able to acquire lock if no other thread has it
  - Thread will block if another thread has the lock (enforces mutual exclusion)
- Lock is released when block terminates
  - End of synchronized block is reached
  - Exit block due to return, continue, break
  - Exception thrown

Fixing Data Race In Our Example



Lock Example

```
public class DataRace extends Thread {
   static int common = 0;
   static Object lockObj = new Object();  // all threads use lockObj's lock

   public void run() {
      synchronized(lockObj) {          // only one thread will be allowed
         int local = common;           // data race eliminated
         local = local + 1;
         common = local;
      }
   }

   public static void main(String[] args) {
      …
   }
}
```

- Keep in mind that lock objects do not need to be static (static is used in the above example to share the lock among all threads)
- How would you solve the data race without using a static lock object? (next slide)

Lock Example (Modified Solution)

```
public class DataRace extends Thread {
   static int common = 0;
   Object lockObj;     // Not static

   public DataRace(Object lockObj) {
      this.lockObj = lockObj;
   }

   public void run() {
      synchronized(lockObj) {          // only one thread will be allowed
         int local = common;           // data race eliminated
         local = local + 1;
         common = local;
      }
   }

   public static void main(String[] args) {
         Object lockObj = new Object();  // all threads use lockObj's lock
         DataRace t1 = new DataRace(lockObj);
         DataRace t2 = new DataRace(lockObj);
         …
   }
}
```

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
- 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
- You must protect the critical section wherever it appears in your code, otherwise several threads may access the critical section simultaneously
  - Protecting the critical section that appears in one part of your code will not automatically protect the critical section everywhere it appears in your code
  - In our example, that translate to having one buyer forgetting to synchronized access to the account. The fact the other buyer is using a lock does not protect the critical section

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
}
```

Synchronized Objects in Java

- Every Java object has a lock
- A lock can be held by only one thread at a time
- A thread acquires the lock by using synchronized
- Acquiring lock example

```
Object x = new Object();  // We can use any object as "locking object"
synchronized(x) {     // try to acquire lock on x on entry
        ...           // hold lock on x in block
}                     // release lock on x on exit
```

- When synchronized is executed
  - Thread will be able to acquire lock if no other thread has it
  - Thread will block if another thread has the lock (enforces mutual exclusion)
- Lock is released when block terminates
  - End of synchronized block is reached
  - Exit block due to return, continue, break
  - Exception thrown

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
- Solution (Example: lockObjInAccount)
  - We are using lockObj to protect access to the Account object
  - What would happen if we define lockObj as static? Can we have multiple accounts?
- 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 lock (e.g., synchronized(this)) then we can rewrite the code by using the synchronized keyword on the method prototype
- Example

```
synchronized foo() { …code… }
// shorthand notation for
foo() {
    synchronized (this) { …code… }
}
```

- Example: synchronizedMethods
- Mutual exclusion for entire body of method

Synchronization Issues

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

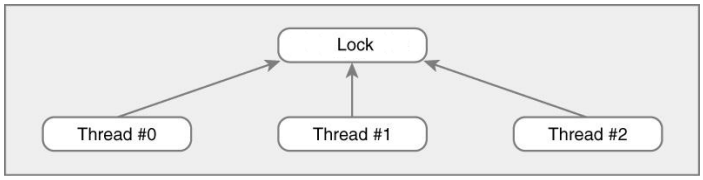
Issue 1- Using Same Lock

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

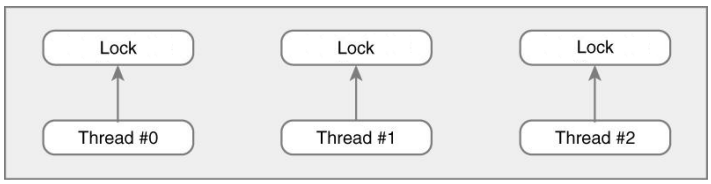
```
foo() {
    Object o = new Object(); // different o per thread
    synchronized(o) {
        … // potential data race
    }
}
```

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;     // all 3 statements must
   local = local + 1;      // be executed together
   common = local;         // 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;
      }                              // transaction not atomic
      synchronized(o) {              // data race may occur
         local = local + 1;          // even using locks
         common = local;
      }
   }
}
```

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 = new Object()
Object b = new Object()
Thread1() {                          Thread2() {
   synchronized(a) {                    synchronized(b) {
      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.”

Miscellaneous

- The lock we have described is known as *intrinsic lock* or *monitor lock*
  - API specification often refers to this entity simply as a "monitor“
- A lock can acquire a lock it already owns (it will not block)
  - Reentrant synchronization
- For a static synchronized method which lock is used?
  - Thread acquires the intrinsic lock for the **Class** object associated with the class
- Reference:
  - http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Synchronization Summary

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