CMSC 132: 
Object-Oriented Programming II

Synchronization in Java

Department of Computer Science
University of Maryland, College Park
Stuff

Coding exercise
- Due Thursday, 6pm
- Will have most of class time today to work on it
- Can be done jointly
  - must note collaborators in submission
- Soon as I get done with lecture notes on synchronization

Demo1 projects
- I'll be asking the top 4 finishers to give a demo to the CMSC 198 class
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

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

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
<th>Thread #3</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{local = common; local = local + 1; common = local;}</code></td>
<td><code>{local = common; local = local + 1; common = local;}</code></td>
<td><code>{local = common; local = local + 1; common = local;}</code></td>
</tr>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
</tr>
<tr>
<td><img src="image4.png" alt="Image" /></td>
<td><img src="image5.png" alt="Image" /></td>
<td><img src="image6.png" alt="Image" /></td>
</tr>
<tr>
<td><img src="image7.png" alt="Image" /></td>
<td><img src="image8.png" alt="Image" /></td>
<td><img src="image9.png" alt="Image" /></td>
</tr>
</tbody>
</table>
Data Race Example

Concurrent execution output (possible case)

Result depends on thread execution order!
public void run() {
    int local = common; // data race
    local = local + 1;
    common = local; // data race
}

public void run() {
    common++; // data race
}

// Both are compiled to identical code
Two Kinds of Problems

- Lack of atomicity
  - developer thinks of a sequence of operations as something that is performed atomically; all at once

- Lack of hand-off
  - compilers and CPU don't like to read/write shared heap locations
  - they try to optimize things, perform updates in registers, local variables, etc
  - Need synchronization to tell them "At this point, you need to be prepared to send/receive data from another thread"
Synchronization

- Used to coordinate communication between threads and provide atomicity
- Incurs runtime overhead
  - Excessive use can reduce performance
  - Generally, only an issue when used excessively in places where threads are not communicating
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
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
  - Additional locks and synchronization added in `java.util.concurrent`

- Example

```java
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

```java
synchronized foo() {  …code…  }
// shorthand notation for
foo() {
    synchronized (this) {  …code…  }
}
```
Synchronized Methods In Java

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

**Shorthand notation for**

```java
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!
- Locks are reentrant: if a thread already owns a lock, it can lock it again
  - no other thread can access it until all locks are released

Lock is always released when block terminates

- End of block reached
- Exit block due to return, continue, break
- Exception thrown
public void run() {
    int local = 0; // Local storage
    // Add one to common
    local = common;
    local = local + 1;
    common = local;
}

Obtain lock for critical section

Release lock

Only one thread can ever be in the critical section
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();
        ...
    }
}
Initially, $x = y = 0$

Thread 1
- $x = 1$
- $a = y$

Thread 2
- $y = 1$
- $b = x$

Can this execution result in $a == 0$ and $b == 0$?
Acquire/Release

- When a thread releases a lock
- and another thread acquire the same lock
- That sets up a happens-before ordering between the release/acquire actions
- Everything done before the release is visible to and ordered before everything done after the acquire
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
  ```java
  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)**
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

```java
synchronized(o) {
    int local = common; // all 3 statements must
    local = local + 1;  // be executed together
    common = local;     // by single thread
}
```
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;
        }
    }
}
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) {
            ...
        }
    }
}

// Thread1 holds lock for a, waits for b

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

// Thread2 holds lock for b, waits for a
void moveMoneyFromBtoA(Bank a, Bank b) {
    synchronized(a) {
        synchronized(b) {
            if (b.x > 0) {
                a.x++; b.x--; // holds lock for a, waits for b
            }
        }
    }
}

Thread1() { mm(a, b); } // holds lock for a, waits for b
Thread2() { mm(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 2\(^{nd}\) lock
- Really really try to avoid holding locks on two objects of the same type, unless you know that they will always be locked in the same order
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 prevent data races
- Java objects support synchronization
- Many other tricky issues
  - To be discussed in future courses