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**

```java
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

```java
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) {
        for (int i = 0; i < 3; i++)
            new DataRace().start();
        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;</code></td>
<td><code>local = common;</code></td>
<td><code>local = common;</code></td>
</tr>
<tr>
<td><code>local = local + 1;</code></td>
<td><code>local = local + 1;</code></td>
<td><code>local = local + 1;</code></td>
</tr>
<tr>
<td><code>common = local;</code></td>
<td><code>common = local;</code></td>
<td><code>common = local;</code></td>
</tr>
</tbody>
</table>

Sequential execution output:

- Thread #1: `local = 0`, `common = 1`
- Thread #2: `local = 1`, `common = 2`
- Thread #3: `local = 2`, `common = 3`
Data Race Example

Concurrent execution output (possible case)

```
<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
<th>Thread #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>local = common;</td>
<td>local = common;</td>
<td>local = common;</td>
</tr>
<tr>
<td>local = local + 1;</td>
<td>local = local + 1;</td>
<td>local = local + 1;</td>
</tr>
<tr>
<td>common = local;</td>
<td>common = local;</td>
<td>common = local;</td>
</tr>
</tbody>
</table>
```

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
- Thread will wait to acquire lock (stop execution)
  - If lock held by another thread
Synchronized Objects in Java

- All (non-Mutable) Java objects provide locks
  - Apply synchronized keyword to object
  - Mutual exclusion for code in synchronization block

Example

```java
Object x = new Object();
synchronized(x) { // acquire lock on x on entry
    ...
    // hold lock on x in block
} // release lock on x on exit
```
Synchronized Methods In Java

- Java methods also provide locks
  - Apply `synchronized` keyword to method
  - Mutual exclusion for entire body of method
  - Synchronizes on object invoking method

Example

```java
synchronized foo() {  // shorthand notation for
    block
    ...code...
}
// shorthand notation for
foo() {
    synchronized (this) {  // block
        ...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!
- Locked block of code ⇒ critical section

Lock is 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

Only one thread can ever be in the critical section

Release lock
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**
  ```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)
Lock Example – Incorrect Version

```java
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
  }
  ```
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;
        }
    }
}
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) {
            ...                   synchronized(b) {
                synchronized(a) {
                    ...

            }               ...
        }
    }
    
}

// Thread1 holds lock for a, waits for b
// Thread2 holds lock for b, waits for a
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
Abstract Data Type – Buffer

- **Buffer**
  - Transfers items from producers to consumers
  - Very useful in multithreaded programs
  - Synchronization needed to prevent multiple consumers removing same item

![Diagram of Producer, Buffer, and Consumer with operations add(item) and remove()]
Buffer Implementation

Class BufferUser() {
    Buffer b = new Buffer();

    ProducerThread() { // produces items
        Object x = new Object();
        b.add(x);
    }

    ConsumerThread() { // consumes items
        Object y;
        y = b.remove();
    }
}
public class Buffer {
    private Object[] myObjects;
    private int numberOfObjects = 0;
    public synchronized add(Object x) {
        myObjects[numberOfObjects++] = x;
    }
    public synchronized Object remove() {
        while (numberOfObjects < 1) {
            // waits for more objects to be added
        } // waits for more objects to be added
        return myObjects[numberOfObjects--];
    }
} // if empty buffer, remove() holds lock and waits
// prevents add() from working ⇒ deadlock
Eliminating Deadlock

```java
class Buffer {
    Object[] myObjects;
    int numberObjects = 0;
    public void add(Object x) {
        synchronized(this) {
            myObjects[numberObjects++] = x;
        }
    }
    public Object remove() {
        while (true) {
            synchronized(this) {
                if (numberObjects > 0) {
                    return myObjects[numberObjects--];
                }
            }
        }
    }
}
```

// if empty buffer, remove() gives up lock
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\textsuperscript{nd} lock
Synchronization Summary

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