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

- **Sequential execution output**

<table>
<thead>
<tr>
<th>Thread</th>
<th>common</th>
<th>local</th>
<th>local + 1</th>
<th>local + 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread #1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread #2</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread #3</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Concurrent execution output (possible case)**

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>common</th>
<th>local</th>
<th>local + 1</th>
<th>local + 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread #2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread #3</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread #1</td>
<td>common</td>
<td>local</td>
<td>local + 1</td>
<td>local + 2</td>
</tr>
<tr>
<td>Thread #2</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread #3</td>
<td></td>
<td>1</td>
<td></td>
<td></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
    - 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
  ```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() {  …code… }
  // shorthand notation for
  foo() {
    synchronized (this) { …code… }
  }
  ```

Synchronized Methods 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
Synchronization Example

```java
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;
            local = local + 1;
            common = local;
        }
    }
    public static void main(String[] args) {
        o = new Object();
    }
}
```

Lock Example

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

```java
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) { // transaction not atomic
            int local = common;
            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**

```java
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
        }
    }
}
```

**Deadlock Example 2**

```java
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

```
Producer
 addItem
 Buffer
 remove()
 Consumer
```

```
```
**Buffer Implementation**

```java
public class Buffer {
    private Object[] myObjects;
    private int numberObjects = 0;

    public synchronized add(Object x) {
        myObjects[numberObjects++] = x;
    }

    public synchronized Object remove() {
        while (numberObjects < 1) {
            // waits for more objects to be added
        }
        return myObjects[--numberObjects];
    }
}
```

→ deadlock

**Eliminating Deadlock**

```java
public class Buffer {
    private Object[] myObjects;
    private int numberObjects = 0;

    public synchronized add(Object x) {
        myObjects[numberObjects++] = x;
    }

    public synchronized Object remove() {
        while (true) {
            synchronized(this) {
                if (numberObjects > 0) {
                    return myObjects[--numberObjects];
                }
            }
        }
    }
}
```

→ if empty buffer, remove() gives up lock

**Synchronization Summary**

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