Synchronization in Java

Fawzi Emad
Chau-Wen Tseng

Department of Computer Science
University of Maryland, College Park

Synchronization Overview

- Data races
- Locks
- Deadlock
- Wait / Notify
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 ThreadExample().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>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

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><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>
<tr>
<td>0</td>
<td>0</td>
<td>0</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
  ...
  // code...
}
// shorthand notation for
foo() {
  synchronized (this) {  // code...
    ...
  }
}
```
Synchronized Methods In Java

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

**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

Locks in Java
Synchronization Example

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();
        ...
    }
}
Synchronization Issues

1. Use same lock to provide mutual exclusion
2. Ensure atomic transactions
3. Avoiding deadlock
4. Use wait & notify to improve efficiency

Issue 1) Using Same Lock

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

Example
```
void foo() {
    Object o = new Object(); // different o per thread
    synchronized(o) {
        ... // potential data race
    }
}
```
Locks in Java

- Single lock for all threads (mutual exclusion)

![Diagram showing a single lock for all threads]

- Separate locks for each thread (no synchronization)

![Diagram showing separate locks for each thread]

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 be executed together
    local = local + 1;            // by single thread
    common = local;
  }
  ```

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) {
            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

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
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();
    }
}

Buffer Implementation

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
        }
        return myObjects[numberOfObjects--];
    }
} // if empty buffer, remove() holds lock and waits
    // prevents add() from working ⇒ deadlock
Eliminating Deadlock

public class Buffer {
    private Object[] myObjects;
    private int numberOfObjects = 0;
    public void add(Object x) {
        synchronized(this) {
            myObjects[numberOfObjects++] = x;
        }
    }
    public Object remove() {
        while (true) {  // waits for more objects to be added
            synchronized(this) {
                if (numberOfObjects > 0) {
                    return myObjects[numberOfObjects--];
                }
            }
        }
        // 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 2nd lock
Issue 4) Using Wait & Notify

Potential problem
- Threads actively waiting for lock consume resources

Solution
- Can wait in queue for lock (in a blocked state) to be notified when lock is released
- Use Thread class methods wait(), notify(), notifyAll()
- Note wait & notify only work when holding a lock

Thread Class Wait & Notify Methods

wait()
- Calling thread is put into blocked state
- Calling thread is placed on wait queue for lock
- Execution continues only if thread reacquires lock

notify()
1. One thread is taken from wait queue for lock
2. Thread is put in runnable state
3. Thread will try to get lock again
4. If thread fails it is put back in blocked state

notifyAll()
- Invokes notify() on all threads in wait queue
Using Wait & Notify

State transitions

Example – Using Wait & Notify

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

    public synchronized add(Object x) {
        myObjects[numberObjects++] = x;
        notify(); // wakes up thread waiting for lock
    }

    public synchronized Object remove() {
        while (numberObjects < 1) {
            wait(); // waits for more objects to be added
        } // will return from wait() only if gets lock
        return myObjects[numberObjects--];
    }
}
```

// if empty buffer, remove() gives up lock via wait()
Example – Cashiers at Checkout

Synchronization Summary

- Needed in multithreaded programs
- Can prevent data races
- Java objects support synchronization
- Many tricky issues