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/update operation
  • Resource → map, set, array, etc.

• Properties
  • Order of accesses may change result of program
  • May cause intermittent errors, very hard to debug
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 thread : allThreads)
            thread.start();
        for (DataRace t : allThreads)
            thread.join();
        System.out.println(common); // May not be 3
    }
}
Data Race Example

- Sequential execution output

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

- Concurrent execution output (possible case)

```plaintext
Thread #1:  local = common;  ➔ 0
Thread #2:  local = common;  ➔ 0
Thread #3:  local = common;  ➔ 0
Thread #1:  local = local + 1;
Thread #2:  local = local + 1;
Thread #3:  local = local + 1;
Thread #1:  common = local;  ➔ 1
Thread #2:  common = local;  ➔ 1
Thread #3:  common = local;  ➔ 1
```
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 used for algorithmic complexity analysis
  - **Thread can acquire/release locks**
  - **Only one thread can acquire lock at a time**
  - **Thread waits to acquire a lock (stops execution) if lock held by another thread**
Java Locks

- 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 (you acquire lock of an object)

```java
Object x = new Object();  // We can use any object as “locking object”
synchronized(x) {
    ...  // Thread tries to acquire lock on x on entry
    ...  // Thread holds lock on x in the block
}  // Thread releases lock on x on exit
```

- When synchronized is executed the:
  - Thread will be able to acquire the 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 is thrown
Fixing Data Race In Our Example

```java
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*
Fixing Previous Example

```java
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 allow the sharing of the lock among all threads)
- How would you solve the data race without using a static lock object? (see next slide)
public class DataRace extends Thread {
    static int common = 0;
    Object lockObj; // Not static lock object reference

    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
  - We don’t need to define an object to protect the Account object as Account object 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 synchronize access to the account. The fact the other buyer is using a lock does not protect the critical section