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/resource, where at least one access is a write
    • 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 t : allThreads)
        t.start();
    for (DataRace t : allThreads)
        t.join();
    System.out.println(common);  // may not be 3
}
Data Race Example

- Sequential execution output

```
Thread #1
{ local = common; ← 0
   local = local + 1;
   common = local; → 1
 }

Thread #2
{ local = common; ← 1
   local = local + 1;
   common = local; → 2
 }

Thread #3
{ local = common; ← 2
   local = local + 1;
   common = local; → 3
 }
```
Data Race Example

- Concurrent execution output (possible case)

```
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 use for algorithmic complexity analysis
  - 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**

- 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
  ```java
  Object x = new Object();  // We can use any object as “locking object”
  synchronized(x) {
    // try to acquire lock on x on entry
    ...
    // hold lock on x in block
  }
  // release lock on x on exit
  ```

- **When synchronized is executed**
  - Thread will be able to acquire 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 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*
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 share the lock among all threads)
- How would you solve the data race without using a static lock object? (next slide)
public class DataRace extends Thread {
    static int common = 0;
    Object lockObj;    // Not static

    public DataRace(Object lockObj) {
        this.lockObj = lockObj;
    }

    public void run() {
        synchronized (lockObj) {
            int local = common;    // only one thread will be allowed
            local = local + 1;    // data race eliminated
            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)
  - Notice we don’t need to define an object to protect the Account object as Account 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 synchronized access to the account. The fact the other buyer is using a lock does not protect the critical section