CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Synchronization in Java II

Department of Computer Science
University of Maryland, College Park
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
  }
  ```
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) {
  // 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
  - 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 thrown
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
• Solution - Example: lockObjInAccount
  • We are using lockObj to protect access to the Account object
  • What would happen if we define lockObj as static? Can we have multiple accounts?
• Solution - Example: usingThisInAccount
  • We don’t need to define an object to protect the Account object as an account object already has a lock
Synchronized Methods In Java

• If the entire body of a method is synchronized using the current object lock (e.g., `synchronized(this)`) we can rewrite the code by using the synchronized keyword on the method prototype

  `synchronized foo() { …code… }`

  // shorthand notation for

  `foo() {`  

  `synchronized (this) { …code… } // this is reference curr object

  }`

• Example: `synchronizedMethods`

• Defining a method as `synchronized` provides mutual exclusion for the entire body of the method

• Only define a method as synchronized if the entire body of the method represents the critical section; otherwise you might limit concurrency
Synchronization Issues

• Use same lock to provide mutual exclusion
• Ensure atomic transactions
• Avoiding deadlock
Issue #1 - Using Same Lock

- Potential problem
  - Mutual exclusion depends on threads acquiring the same lock
  - No synchronization will take place if threads use different locks

- Example
  ```java
  foo() {
    Object o = new Object(); // Different object (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)
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;
        }
    }
    public static void main(String[] args) {
        ...
    }
}
Issue #2 - Atomic Transactions

• Potential problem
  • Sequence of actions representing the critical section must be performed as single atomic transaction to avoid a data race
  • We need to ensure lock is held for the duration of the execution of the critical section
    • We cannot perform part of the instructions representing the critical section, release the lock, acquire the lock again, and complete the rest of the instructions

• Example
  synchronized(o) {
    int local = common; // All 3 statements must be executed
    local = local + 1; // together while holding the lock
    common = local;
  }
public class DataRace extends Thread {
    static int common = 0;
    static Object o = new Object(); // All threads use o’s lock
    public void run() {
        int local;

        synchronized(o) {
            local = common;
        }
        synchronized(o) {
            // Transaction is not atomic
            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 \(a\) = new Object()
Object \(b\) = new Object()

Thread1() {
    synchronized(a) {
        synchronized(b) {
            ...
        }
    }
}

Thread2() {
    synchronized(b) {
        synchronized(a) {
            ...
        }
    }
}

// Thread1 holds lock for a, waits for b
// 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
Deadlock Example 3

- When two friends bow to each other at the same time
- [https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html](https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html)
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
Thread-safe

- Thread-safe - Code is considered thread-safe if it works correctly when executed by multiple threads simultaneously
- **Example:** ArrayList is not thread-safe
  From Java API: “Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally.”
Miscellaneous

- The lock we have described is known as *intrinsic lock* or *monitor lock*
  - API specification often refers to this entity simply as a "monitor"
- A thread can acquire a lock it already owns (it will not block)
  - Reentrant synchronization
- For a static synchronized method which lock is used?
  - Thread acquires the intrinsic lock for the **Class** object associated with the class
- Reference:
  - [http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html](http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html)
Designing Solutions

- You must be careful while designing solutions involving threads
- **Make sure you are not limiting concurrency**
  - Correctly identify what represents the critical section
  - Avoid unnecessary synchronization
- Test
  - Make sure you test your code to identify performance issues and data races
Synchronization Summary

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