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, 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
  }
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
\[
\begin{align*}
\text{local} &= \text{common}; & 0 \\
\text{local} &= \text{local} + 1; \\
\text{common} &= \text{local}; & 1
\end{align*}
\]

Thread #2
\[
\begin{align*}
\text{local} &= \text{common}; & 1 \\
\text{local} &= \text{local} + 1; \\
\text{common} &= \text{local}; & 2
\end{align*}
\]

Thread #3
\[
\begin{align*}
\text{local} &= \text{common}; & 2 \\
\text{local} &= \text{local} + 1; \\
\text{common} &= \text{local}; & 3
\end{align*}
\]
Data Race Example

- Concurrent execution output (possible case)
**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
  
  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;
}
```

1. **Obtain lock for critical section**
2. **Release lock**

*Only one thread can ever be in the critical section*
Lock 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 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) {
        lockObj = this.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 using 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