Sharing Objects – Ch. 3

- Visibility
- What is the source of the issue?
- Volatile
- Dekker's algorithm
- Publication and Escape
- Thread Confinement
- Immutability
- Techniques of safe publication
- Assignment

Visibility

- To write correct shared-state concurrent programs we have to know *when* changes made in one thread become visible in other threads
- Our intuitions gained from sequential programming provide the *wrong answers!*

Sequential Consistency

- Language definition assures you that if you assign to a variable in one statement, the effect of that assignment will be visible in a later-executed statement

  ```
  x = 1;
  x = x+1;
  if (x==2) { … }
  ```

- *But* only if the statements are executed in the same thread!
- Think how awful programming would be if this were not the case!

Sequential consistency does not hold for concurrent threads

- **An assignment may *never* become visible in a different thread**
- **Assignments done in some order in one thread may become visible in arbitrary order in a different thread and in different orders in different threads**

y = 0; x = 1; x = 3; y = 2;

//

if (y==2) { // can't assume that x==3 here }

Not even that x==1 || x == 3

Why?

- Compiler writers and computer architects pursue speed in the usual case
    - Keep variables in registers as much as possible
    - Re-order stores to exploit memory architecture
    - Re-order instructions, move them out of loops, etc. to improve performance
- These optimizations operate without knowledge of any concurrent activity (esp. the hardware ones).

How do we fix this?

- Synchronization
- Control of visibility is a second role for synchronization – the first was to provide atomicity
- The same mechanisms that provide atomicity also fix the visibility problem
  - Another synchronization mechanism called “volatile” fixes visibility but not atomicity
- “stale” data is possible unless synchronization is used for *every* access, read *and* write, to a variable

Out-of-thin-air safety

- Even if you don’t use synchronization for shared variable accesses Java guarantees that what is read will be something that was written by your program (or automatically initialized)
- EXCEPTION: 64-bit *longs* and *doubles*
- NOTE: C/C++ do not make this guarantee even for sequential code
- (Your program will not see values that appear out of thin air)

Visibility Guarantee Provided by Intrinsic Locks

- A thread holding a lock is guaranteed to see all updates performed while any other thread previously held the same lock.
- Another reason for the rule: “every shared variable should be protected by exactly one lock”

volatile variables

- **Any data member variable or static variable can be declared volatile**

  volatile int x;

- **Accesses to volatile variables require no locking and hence cannot block**
- **After writing a volatile variable x in thread A and reading it in thread B, thread B can see *all* writes visible to A at the time of its write, not just the write to x**

Using a volatile variable instead of locks

- Writes to the variable do not depend on its previous value *or* the variable is only updated in one thread
- The variable is not related by an invariant to other shared variables
- Locking is not needed for any other reason (if locking is used, volatile is unnecessary)

Terminology and History

- A *critical section* is a general term for code sequence that must be executed atomically for correctness.
- Synchronized blocks implement critical sections
- Before hardware implementations had explicit synchronization instructions (test-and-set, e.g.) programmers had to protect critical sections using only normal memory reads and writes

Dekker’s synchronization algorithm

```
boolean enter1 = false;
boolean enter2 = false;
int turn = 1;
{ while(true) { /* Thread 1 */
   enter1 = true;
   while (enter2) {
      if (turn==2) {
         enter1 = false;
         while (turn==2) yield();
         enter1 = true;
      }
   }
   /* critical section */
   enter1 = false; turn = 2;
   /* non-critical section */
}}
```

```
/* Thread 2 */
{ while(true) {
   enter2 = true;
   while (enter1) {
      if (turn==1) {
         enter2 = false;
         while (turn==1) yield();
         enter2 = true;
      }
   }
   /* critical section */
   enter2 = false; turn = 1;
   /* non-critical section */
}}
```

Discussion

- What has to be done to Dekker’s algorithm in light of our previous discussion about visibility?
- Like other manually constructed synchronization techniques, Dekker’s algorithm is intended to:

1. Provide mutual exclusion
2. Avoid deadlock
3. Avoid unnecessary delay – if one thread wants in and the other doesn’t the first is not delayed
4. Ensure eventual entry – if a thread wants in it eventually gets in

Assignment – Please, no handwritten work

1. After inserting the necessary volatile declarations, argue convincingly that Dekker's algorithm exhibits the four properties listed on slide 13.
2. Based on what you know so far, how well does Java's intrinsic synchronization meet these properties
3. Write sequential code that abuses the class UnsafeStates in Fig. 3.6.
4. Turn in on web site turnin page by Jan. 31.

Publication – part 1: avoiding escape

- Publishing – making an object available outside of its current scope
  - Store it where other code can find it
  - Return it from a non-private method
  - Pass it to a method of another class
- Escape – incorrect publication
  - Publishing internal, private state (violates encapsulation)
  - Publishing an object also publishes objects referenced by its non-private fields
  - Publishing an object to a different thread, while it is being constructed, violates thread safety

Unsafe approach to listener registration – Fig. 3.7

```
public class ThisEscape {
  public ThisEscape(EventSource source) {
    source.registerListener(
      new EventListener() {
        public void onEvent(Event e) {
          doSomething(e);
        }
      });
  }
}
```

SafeListener – Fig. 3.8

```
public class SafeListener {
   private final EventListener listener;
   private SafeListener () {
      listener = new EventListener() {
         public void onEvent(Event e) {
            doSomething(e);
         }
      };
   }
   public static SafeListener newInstance(EventSource source) {
      SafeListener safe = new SafeListener();
      source.registerListener(safe.listener);
      return safe;
   }
}
```

2. Thread Confinement

- Recall that one approach to thread safety is to not share state between threads
- How can we do that:

1. Only ever put object reference on the stack (in local variables) – relies on the property of Java that references to stack variables cannot be obtained.
2. Use the ThreadLocal class: it’s getter and setter store values s.t. each thread has its own copy
3. Ad hoc thread confinement

3. Immutability

- How to do immutability properly is itself a bit tricky – next time.

Publication Part 2: Safe publication

- **Previously: how to avoid unwanted publication**
- **Now: how to safely publish when publication is desired**