**Lecture 10: Java Synchronization; Scope**

Week 6, Tuesday, May 7, 2019  
  
**What’s bad about race conditions?**

* Nondeterminism (hard to debug)
* Crashes
* Misbehavior

**Java synchronization**

* synchronized keyword
  + Use around small critical sections (“monitors”)
    - Prohibit races where two threads access the same object simultaneously
  + Honor system: hidden failure if it’s not used correctly
  + Simple, but have performance problems when contention is high
* wait/notify/notifyAll: recall from last lecture
* Higher-level synchronization library: java.util.concurrent
  + Semaphore
    - Size: number of total resources
    - acquire(): tries to decrease available resources; waits if busy
    - tryAcquire(): tries to decrease available resources; returns false if busy
    - release(): increases available resources
  + Exchanger
    - Analogy: spy and spymaster set up rendezvous point
    - Exchanger x = …;
    - t1 (spymaster): t2 (spy):

intelligence = x.exchange(moneyFromBank); money = x.exchange(secret);

* + - If the other side is not yet ready, exchange() waits
  + CountDownLatch
    - Wait until all threads finish, do some work, and then restart all threads
    - Analogy: Kentucky derby: wait until all horses/jockeys are ready, and then open the gates
  + CyclicBarrier
    - Like CountDownLatch, but reusable
    - Example: weather simulation; after a set interval of parallel computation, exchange information, and restart simulation

**Language implementation optimization**

* Implementation of synchronized has evolved over time: spinlocks → speculative execution, better CPU primitives
* “As-if” rule
  + Language is specified using a specific model. But:
  + Language implementation can be done any way one would like, so long as the resultant behavior “as if” the model is executed.
* Example: *L* standard says: the memory is laid out in a little-endian fashion
  + But on a big-endian machine, the compiler can translate code to big-endian instructions
* More extreme examples:

static char a[8] = "abcdefgh";

* + - The compiler could lay out a in non-contiguous memory locations

int c = 10; int b = 20;

* + - mov $20, 40(%rsp)

mov $10, 44(%rsp)

* + - The compiler can reorder instructions so long as the difference is not observable
* In computer science, “optimization” does not guarantee faster execution; freedom for compilers to try
* Optimizations are key to achieve higher performance, but may hinder correct multithreading

  + p->l = 1;
  + p->payload = sqrt(…);

p->l = 0;

* + - In C, the first assignment to p->l may be removed (“optimized away”).
    - However, between the assignments to p->l, the programmer may intend for another thread to access p->l; the as-if rule is violated under this optimization.
* Optimizations are commonly done under a set of assumptions. In C, the assumption is single-threaded.
* Annotations may help
  + volatile keyword (volatile int x;)
    - Advises compiler not to optimize away certain memory accesses or reorder accesses
    - In C and C++, volatile ≠ atomic
      * Atomic: ensure reads/writes are not teared (mish-mash of old/new values)
    - In Java, volatile primitives = atomic

**Java Memory Model**

* Problem with synchronized keyword had JMM not existed

  + class C {
  + int n;
  + synchronized int incr() {
  + return n++;
  + }

}

* + Potential assembly code:
  + incr:
  + LOCK
  + movl (%rbp), %rax
  + addl $1, %rax
  + movl %rax, (%rbp)
  + subl $1, %rax // fix return value
  + UNLOCK

ret

Compiler can notice that LOCK/UNLOCK does not access (%rbp), and reorder UNLOCK

incr:

LOCK

UNLOCK

…

Obviously bogus

* JMM specify rules for when the compiler can reorder operations
* Operations of JVM
  + (A) Normal load; normal store: Optimize any way!
  + (B) Volatile load; enter monitor: Still optimize, but be careful
  + (C) Volatile store; exit monitor: Most careful, as we can now change the variable
* Reordering rules:
  + Can we reorder A1A2BA3 → A2A1BA3?
  + Always: must not reorder if behavior of sequential execution is changed
    - Always not allowed: `x=1; x=2; → x=2; x=1;`
  + Allowed interchange table
  + 1\2 | A | B | C
  + A | Y | Y |
  + B | | |

C | Y | |

(Vertical: first operation; horizontal: second operation)

* + - Example: int x, y; volatile int w;
    - A-A: x=1; y=2; → y=2; x=1;
    - A-B: x=1; w; → w; x=1; (moving normal op down into critical section)
    - C-A: w=1; x=2; → x=2; w=1; (moving normal op up into critical section)

**Scope**

* The **scope** of an identifier as the set of program locations where the identifier is visible
* Block scope
  + {1 int n1 = …;  
    {2 int n2 = …; n2; }2 }1
  + In {2…}2, the outer n1 is not visible; n1 still exists though. Scope ≠ lifetime
* In most languages, scope is static, but lifetime is a runtime notion
* The process of associating a value to a name is called **binding**
* Questions for a particular language: what kind of value may be bound to certain names? when is the “binding time”?