WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Lecture 8 – Ch 5

- Higher-level concurrency patterns
- Synchronized collections
- Concurrent collections
- Producer-consumer pattern
  - Serial thread confinement
- Interruptable waiting
- Synchronizers
- Concurrent result cache

Synchronized Collections (5.1)

- Collections are thread-safe for their own invariants
- Client invariants are nevertheless at risk
- Requirement for client-side locking
  - Compound actions: find, put-if-absent, iterate
  - Locking policy: intrinsic lock on the collection
  - Locks held too long limit performance
- ConcurrentModificationException
  - Fail-fast
  - Client finds out something went wrong, has to deal with it

Concurrent Collections (5.2)

- Evolution of libraries based on experience
- Finer-grained locking (not on the whole collection – lock striping)
- Concurrent reads; partially concurrent writes
- Weakly-consistent iterators
  - Return all items in the collection at the time iteration started and maybe some that were later added
- Some useful compound actions provided (client does not have to build them – can't)
- Commonly used collection is the ConcurrentHashMap<K,V>
  - Put-if-absent; remove-if-equal; replace-if-equal

Lock Striping (11.4.3)

- Have a lock for independent subsets of a collection (e.g. HashMap): N hash buckets N/16 locks.
- Issue: how do you protect the whole collection? Acquire ALL of the locks (recursively!)

Lock striping

Another concurrent collection

- CopyOnWriteArray{List,Set}
  - Copying cost – but good when collection is seldom updated
  - “Purely functional data structures” by Chris Okasaki

Producer-consumer pattern

- Originally a program structuring convenience; now one way to achieve parallelism
- Similar to shell pipes – which just transfer streams of bytes
  - But data objects are typed and structured
- Visitor pattern
  - For each item in a collection perform an action
  - Ex: Treewalk – the tree walker has state for keeping track of where it is
  - What if the action involves keeping complicated state? - compression
- Producer-consumer pattern lets both the producer and consumer keep state from one item to the next

Producer-Consumer Boundary: Blocking Queue

- Finite capacity
  - Java also has unbounded queues – use is not recommended
- put() waits if full
- get() waits if empty

```
class SimpleBlockingQueue<T> {
  private T[] rep;
  private int head, tail, size;
  public BQ(int size) {
    rep = new T[size];
    head = 0; tail = 0; this.size=size;
  }
  public synchronized T take() {
    T result;
    while (head==tail) wait();
    result = rep[head];
    head += 1; // mod size
    notify();
  }
  public synchronized put(T elem) {
    while (tail+1==head) wait(); // mod size
    rep[tail] = elem;
    tail += 1; // mod size
    notify();
  }
}
```

- Are the empty and full conditions correct?
- Does it work if size==1? (Queue with size==1 is often called a *mailbox*
- How could you allow concurrent putting and taking? What would be the problem cases?

Extended blocking queue interface

- offer and poll methods
  - Don't wait – always return immediately
  - Allow clients to decide what to do if blocking would occur
  - Note: have to be careful using these in check-act situations. Why?

Serial thread confinement

- Recall thread confinement – object is only accessible from a single thread
- An object only accessed from a single thread does not require synchronization
- Serial thread confinement – after passing an object into a blocking queue the producer never touches it again. The synchronization of the blocking queue suffices to safely publish the object to the consumer
- Other mechanisms for safe publishing can also be used in the serial confinement pattern

Deques

- Double-ended queues
- Pronounced “deck”
- Put and take at either end

Cooperative Interruption

- Interruptions occur only when a thread is blocked
  - Interruption is delivered as a InterruptedException
- This is a nice model
- Thread is free to swallow an interrupted exception (not good practice)
- Contrast with C signals and signal handlers
  - Signal handler can start executing any time the signal is not blocked
  - This is not a nice model

Don’t swallow InterruptedException (Fig. 5.10)

- Methods that call things that can raise InterruptedException must either
  - Be declared as throw’ing InterruptedException, or
  - Catch InterruptedException
    - Clean up local state
    - Thread.currentThread.interrupt() // restore interrupt status if InterruptedException cannot be re-raised
  - Or both
    - Re-raising InterruptedException after cleaning up local state

Other fun synchronizers - Latches

- CountDownLatch – initial non-zero value
- Blocks threads calling latch.await() until latch value is 0; once 0 value never changes – "sticky" "monotonic property"
- latch.countDown() reduces the latch value
- Co-ordinated starting and ending points
  - Create N tasks that wait on a startingLatch (value 1)
  - startingLatch.countDown()
  - endingLatch.await() // endingLatch initial value N
  - Each of N threads calls endingLatch.countDown()

FutureTask

- Think of it as a thread body that computes a result value

- th = new Thread(future); // different constructor than Thread(runnable)
- future.get() returns the result of the execution – complicated by how to handle exceptions. Not very elegant to my mind

Semaphores

- Like latches, an integer value
- acquire(): if value is 0 wait, otherwise reduce the value by 1
- release(): increase the value by 1

Barriers

- CyclicBarrier – allow N threads to repeatedly all gather at the barrier
- Think about rewriting the example used in Hwk1 to use latches and/or barriers in place of the homegrown barriers that I used.

Results cache (5.6)

- Very interesting example
- Sometimes called a “memo” cache
- Illustrates implementation of caching – a common technique to increase performance
  - Idea: if it takes a long time to get an answer, save it for awhile in case you need it again
    - Memory caches, disk caches, results caches

Stages in Cache Design

- Cache for concurrent use is tricky
  - Obvious, use intrinsic lock around everything. Problem: lock held during long computation limits concurrency
  - Use ConcurrentHashMap – better concurrency but unsynchronized check-then-act with long "act" potentially allows unneeded work
  - Cache a FutureTask in the ConcurrentHashMap – still unsynchronized check-then-act but now the "act" is just inserting a FutureTask rather than the long expensive computation
  - Use putIfAbsent to eliminate this final race