Parallel & Concurrent Programming:
Advanced Java Concurrency

Emery Berger
CMPSCI 691W - Spring 2006
Outline

- Last time:
  - Built-in Java concurrency
    - Thread, synchronized, wait(), notify()...
  - New in java.util.concurrent
    - Semaphores
    - Blocking queues, barriers, futures

- Today:
  - Lock “improvements”
  - Non-blocking operations
  - java.nio library
Problems with Locks

- No way to
  - Try to acquire lock
  - Give up after timeout
  - Use reader/writer locking
- Locks always reentrant
- Access control:
  - Any method can call `synchronized(obj)`
  - Only block-structured locking
- Locks may `block`
New Lock classes

- java.util.concurrent.locks
  - Familiar Lock interface
    - lock(), unlock()
    - tryLock()
    - tryLock(time, unit)
  - ReentrantLock
  - ReentrantReadWriteLock

- Support for rolling your own
  - Condition
class BoundedBuffer {
    final Lock lock =
        new ReentrantLock();
    final Condition notFull =
        lock.newCondition();
    final Condition notEmpty =
        lock.newCondition();

    public void put (Object x)
        throws InterruptedException {
            lock.lock();
            try {
                while (count == 0)
                    notEmpty.await();
                items[putptr] = x;
                if (++putptr == items.length)
                    putptr = 0;
                ++count;
                notFull.signal();
            } finally { lock.unlock(); }
        }

    public Object take()
        throws InterruptedException {
            lock.lock();
            try {
                while (count == items.length)
                    notEmpty.await();
                Object x = items[takeptr];
                if (++takeptr == items.length)
                    takeptr = 0;
                --count;
                notFull.signal();
                return x;
            } finally { lock.unlock(); }
        }
}
Non-blocking Atomics

- Locks can block ⇒
  - Priority inversion
  - Can wait unbounded time till success
  - Deadlock, relatively slow, convoying
- java.util.concurrent.atomic
  - Provides access to hardware-level atomic operations
  - Building blocks for non-blocking data structures
Non-blocking Atomics

- **AtomicInteger**
  - `set(int)`
  - `get()`
  - `addAndGet(int)`, `incrementAndGet()`
  - `getAndAdd(int)`, `getAndIncrement()`
  - `compareAndSet(expected, update)`
    - Atomically sets value to updated value iff current value == expected value
    - True iff successful
- No locks used on most platforms
Non-blocking Atomics

- AtomicReference\(\langle V\rangle\)
  - `set(V newValue)`
  - `get()`
  - `compareAndSet(expected, update)`
  - `getAndSet(newValue)`
    - Returns old value
Non-blocking stack

- push (N)
  - h = Head; N->next = h
  - Repeat until
    CAS(Head, h, N)
- pop:
  - h = Head;
    next = h->next
  - Repeat until
    CAS(head, h, next)
  - Return h
- But: ABA problem
Non-blocking stack

- push (N)
  - h = Head; N->next = h
  - Repeat until
    CAS(Head, h, N)
- pop:
  - h = Head;
  - next = h->next
  - Repeat until
    CAS(head, h, next)
  - Return h
- But: ABA problem
Non-blocking stack

- push (N)
  - h = Head; N->next = h
  - Repeat until
    CAS(Head, h, N)
- pop:
  - h = Head;
    next = h->next
  - Repeat until
    CAS(head, h, next)
  - Return h
- But: ABA problem
Non-blocking stack

- push (N)
  - h = Head; N->next = h
  - Repeat until
    CAS(Head, h, N)
- pop:
  - h = Head;
    next = h->next
  - Repeat until
    CAS(head, h, next)
  - Return h
- But: ABA problem
Non-blocking stack

- push (N)
  - h = Head; N->next = h
  - Repeat until
    CAS(Head, h, N)
- pop:
  - h = Head;
    next = h->next
  - Repeat until
    CAS(head, h, next)
  - Return h
- But: ABA problem
Non-blocking stack

- push (N)
  - h = Head; N->next = h
  - Repeat until CAS(Head, h, N)
- pop:
  - h = Head;
    next = h->next
  - Repeat until CAS(head, h, next)
  - Return h
- But: ABA problem
Non-blocking stack

- push (N)
  - h = Head; N->next = h
  - Repeat until CAS(Head, h, N)
- pop:
  - h = Head;
  - next = h->next
  - Repeat until CAS(head, h, next)
  - Return h
- But: ABA problem
Versioning

- One solution – use **tags**
  - Associate **version number** with refs
- `AtomicStampedReference<V>`
  - `set(V newReference, int newStamp)`
  - `get(stampHolder)`
  - `compareAndSet(expectedRef, newRef, expectedStamp, newStamp)`

- Non-blocking only on supporting architectures
  - x86, but not 64-bit
Non-blocking stack

- push (N)
  - h = Head; N->next = h
  - Repeat until
    CAS(Head, h, N)
- pop:
  - h = Head;
  - next = h->next
  - Repeat until
    CAS(head, h, next)
  - Return h
Non-blocking stack

- push (N)
  - \( h = \text{Head}; \ N \rightarrow \text{next} = h \)
  - Repeat until \( \text{CAS(Head, h, N)} \)
- pop:
  - \( h = \text{Head}; \)
  - next = h->next
  - Repeat until \( \text{CAS(head, h, next)} \)
  - Return h
- ABA problem solved
Java (as of 1.4) supports **non-blocking I/O** and other low-level I/O

- Memory mapped byte buffers
- Channels
- Pipes
- Selectors
Memory-mapped buffers

- Array mapped to file on disk
  - Uses virtual memory operations
- Access to buffer in memory = file operation
- Much faster than direct calls to file I/O
- Why?
Selector

- Essentially same notion as `select()`
- Add any channels of interest and start I/O operations
  - Must have configured as non-blocking: `sc.configureBlocking(false)`
- Returns iterator to channels ready for I/O operations
The End

- No class next week
- Homework due Feb 27
- Next time(s):
  server architectures, SEDA, Flux
  - Read SEDA & Flux papers