The Perils of Concurrency, Part 2

Can't live with it.
Can't live without it.

Jonathan Aldrich

Charlie Garrod
• Homework 5: The Framework Strikes Back
  ▪ You should have already selected a partner(s)
  ▪ 5a due at your scheduled time next Wednesday
  ▪ Commit/push designs by 8:59 a.m.
Key topics from Tuesday
Last time: Concurrency, part 1

- The concurrency backstory
  - Motivation, goals, problems, ...

Problems of concurrency

- Realizing the potential
  - Keeping all threads busy doing useful work

- Delivering the right language abstractions
  - How do programmers think about concurrency?
  - Aside: parallelism vs. concurrency

- Non-determinism
  - Repeating the same input can yield different results
Last time: Concurrency, part 1

- The concurrency backstory
  - Motivation, goals, problems, ...

http://www.genome.gov/sequencingcosts/

Problems of concurrency

- Realizing the potential
  - Keeping all threads busy doing useful work

- Delivering the right language abstractions
  - How do programmers think about concurrency?
  - Aside: parallelism vs. concurrency

Same input can yield different results
Bad news: some simple actions are not atomic

• Consider a single 64-bit `long` value

  high bits  low bits

  ▪ Concurrently:
    • Thread A writing high bits and low bits
    • Thread B reading high bits and low bits

Precondition:

```
long i = 10000000000;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

```
ans: 01001...00000000
(10000000000)
```

```
ans: 00000...00101010
(42)
```

```
ans: 01001...00101010
(10000000042 or ...)
```
Today: Concurrency, part 2

• Race conditions, revisited

• Primitive concurrency in Java
  ▪ Explicit synchronization with threads and shared memory
  ▪ More concurrency problems

• Higher-level abstractions for concurrency (still mostly not today)
  ▪ Data structures
  ▪ Higher-level languages and frameworks
  ▪ Hybrid approaches
Race conditions in real life

- E.g., check-then-act on the highway
Race conditions in your life

- E.g., check-then-act in simple code

```java
public class StringConverter {
    private Object o;
    public void set(Object o) {
        this.o = o;
    }
    public String get() {
        if (o == null) return "null";
        return o.toString();
    }
}
```

- See StringConverter.java, Getter.java, Setter.java
Primitive concurrency control in Java

• Each Java object has an associated intrinsic lock
  ▪ All locks are initially unowned
  ▪ Each lock is exclusive: it can be owned by at most one thread at a time

• The synchronized keyword forces the current thread to obtain an object's intrinsic lock
  ▪ E.g.,

        synchronized void foo() { ... } // locks "this"

        synchronized(fromAcct) {
            if (fromAcct.getBalance() >= 30) {
                toAcct.deposit(30);
                fromAcct.withdrawal(30);
            }
        }

• See SynchronizedIncrementTest.java
Primitive concurrency control in Java

- `java.lang.Object` allows some coordination via the intrinsic lock:
  ```java
  void wait();
  void wait(long timeout);
  void wait(long timeout, int nanos);
  void notify();
  void notifyAll();
  ```

- See `Blocker.java`, `Notifier.java`, `NotifyExample.java`
Primitive concurrency control in Java

- Each lock can be owned by only one thread at a time
- Locks are re-entrant: If a thread owns a lock, it can lock the lock multiple times
- A thread can own multiple locks

```java
synchronized(lock1) {
    // do stuff that requires lock1

    synchronized(lock2) {
        // do stuff that requires both locks
    }

    // ...
}
```
Another concurrency problem: deadlock

- E.g., Alice and Bob, unaware of each other, both need file $A$ and network connection $B$
  - Alice gets lock for file $A$
  - Bob gets lock for network connection $B$
  - Alice tries to get lock for network connection $B$, and waits...
  - Bob tries to get lock for file $A$, and waits...

- See Counter.java and DeadlockExample.java
Dealing with deadlock (abstractly, not with Java)

• Detect deadlock
  § Statically?
  § Dynamically at run time?

• Avoid deadlock

• Alternative approaches
  § Automatic restarts
  § Optimistic concurrency control
Detecting deadlock with the waits-for graph

- The *waits-for graph* represents dependencies between threads
  - Each node in the graph represents a thread
  - A directed edge T1->T2 represents that thread T1 is waiting for a lock that T2 owns
- Deadlock has occurred iff the waits-for graph contains a cycle
- Got a problem with this?
Deadlock avoidance algorithms

- Prevent deadlock instead of detecting it
  - E.g., impose total order on all locks, require locks acquisition to satisfy that order
  - Thread:
    - acquire(lock1)
    - acquire(lock2)
    - acquire(lock9)
    - acquire(lock42) // now can't acquire lock30, etc...

- Got a problem with this?
Avoiding deadlock with restarts

• One option: If thread needs a lock out of order, restart the thread
  ▪ Get the new lock in order this time

• Another option: Arbitrarily kill and restart long-running threads
Avoiding deadlock with restarts

- One option: If thread needs a lock out of order, restart the thread
  - Get the new lock in order this time

- Another option: Arbitrarily kill and restart long-running threads

- Optimistic concurrency control
  - e.g., with a copy-on-write system
  - Don't lock, just detect conflicts later
    - Restart a thread if a conflict occurs
Another concurrency problem: livelock

- In systems involving restarts, *livelock* can occur
  - Lack of progress due to repeated restarts
- *Starvation*: when some task(s) is(are) repeatedly restarted because of other tasks
Concurrent control in Java

• Using primitive synchronization, you are responsible for correctness:
  ▪ Avoiding race conditions
  ▪ Progress (avoiding deadlock)

• Java provides tools to help:
  ▪ volatile fields
  ▪ java.util.concurrent.atomic
  ▪ java.util.concurrent
The Java *happens-before* relation

- Java guarantees a transitive, consistent order for some memory accesses
  - Within a thread, one action *happens-before* another action based on the usual program execution order
  - Release of a lock *happens-before* acquisition of the same lock
  - `Object.notify` *happens-before* `Object.wait` returns
  - `Thread.start` *happens-before* any action of the started thread
  - Write to a *volatile* field *happens-before* any subsequent read of the same field
  - ...

- Assures ordering of reads and writes
  - A race condition can occur when reads and writes are not ordered by the happens-before relation
The `java.util.concurrent.atomic` package

- **Concrete classes supporting atomic operations**
  - `AtomicInteger`
    - `int get();`
    - `void set(int newValue);`
    - `int getAndSet(int newValue);`
    - `int getAndAdd(int delta);`
    - ...
  - `AtomicIntegerArray`
  - `AtomicBoolean`
  - `AtomicLong`
  - ...

  ![Image](image-url)
The `java.util.concurrent` package

- Interfaces and concrete thread-safe data structure implementations
  - ConcurrentHashMap
  - BlockingQueue
    - ArrayBlockingQueue
    - SynchronousQueue
  - CopyOnWriteArrayList
  - ...

- Other tools for high-performance multi-threading
  - ThreadPools and Executor services
  - Locks and Latches
java.util.concurrent.ConcurrentHashMap

• **Implements** java.util.Map<K,V>
  - High concurrency lock striping
    • Internally uses multiple locks, each dedicated to a region of the hash table
    • Locks just the part of the table you actually use
    • You use the ConcurrentHashMap like any other map...

| Locks | Hashtable | Locks |

15-214 Garrod
java.util.concurrent.BlockingQueue

• Implements java.util.Queue<E>

• java.util.concurrent.SynchronousQueue
  ▪ Each `put` directly waits for a corresponding `poll`
  ▪ Internally uses `wait/notify`

• java.util.concurrent.ArrayBlockingQueue
  ▪ `put` blocks if the queue is full
  ▪ `poll` blocks if the queue is empty
  ▪ Internally uses `wait/notify`
The CopyOnWriteArrayList

- Implements `java.util.List<E>`
- All writes to the list copy the array storing the list elements
Next week:

• More concurrency