Principles of Software Construction: Objects, Design and Concurrency

The Perils of Concurrency, part 2

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

Jonathan Aldrich        Charlie Garrod

© 2012 C Garrod, J Aldrich, and W Scherlis
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
Today: Concurrency, part 2

• 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
Basic concurrency in Java

• The `java.lang.Runnable` interface
  ```java
  void run();
  ```

• The `java.lang.Thread` class
  ```java
  Thread(Runnable r);
  void start();
  static void sleep(long millis);
  void join();
  boolean isAlive();
  static Thread currentThread();
  ```

• See `IncrementTest.java`
Atomicity

• An action is \textit{atomic} if it is indivisible
  ▪ Effectively, it happens all at once
    • No effects of the action are visible until it is complete
    • No other actions have an effect during the action

• In Java, integer increment is not atomic

\begin{itemize}
  \item \texttt{i++;}
  \item is actually
  \begin{enumerate}
    \item Load data from variable \texttt{i}
    \item Increment data by 1
    \item Store data to variable \texttt{i}
  \end{enumerate}
\end{itemize}
One concurrency problem: race conditions

- A race condition is when multiple threads access shared data and unexpected results occur depending on the order of their actions.

- E.g., from IncrementTest.java:
  - Suppose `classData` starts with the value 41:

    Thread A:
    ```java
    classData++;  
    ```

    Thread B:
    ```java
    classData++;  
    ```

    One possible interleaving of actions:
    1A. Load data(41) from `classData`
    1B. Load data(41) from `classData`
    2A. Increment data(41) by 1 -> 42
    2B. Increment data(41) by 1 -> 42
    3A. Store data(42) to `classData`
    3B. Store data(42) to `classData`
Race conditions in real life

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

- E.g., check-then-act at the bank
  - The "debit-credit problem"

**Alice, Bob, Bill, and the Bank**

- A. Alice to pay Bob $30
  - Bank actions
    1. Does Alice have $30?
    2. Give $30 to Bob
    3. Take $30 from Alice

- B. Alice to pay Bill $30
  - Bank actions
    1. Does Alice have $30?
    2. Give $30 to Bill
    3. Take $30 from Alice

- If Alice starts with $40, can Bob and Bill both get $30?
Race conditions in real life

- E.g., check-then-act at the bank
  - The "debit-credit problem"

**Alice, Bob, Bill, and the Bank**

- **A. Alice to pay Bob $30**
  - Bank actions
    1. Does Alice have $30?
    2. Give $30 to Bob
    3. Take $30 from Alice

- **B. Alice to pay Bill $30**
  - Bank actions
    1. Does Alice have $30?
    2. Give $30 to Bill
    3. Take $30 from Alice

- **If Alice starts with $40, can Bob and Bill both get $30?**

A.1
A.2
B.1
B.2
A.3
B.3!
Race conditions in *your* real 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
Some actions are atomic

Thread A:

```java
int i = 7;
i = 42;
```

Thread B:

```java
ans = i;
```

- What are the possible values for ans?
Some actions are atomic

Thread A:
\[
\begin{align*}
\text{int } i &= 7; \\
i &= 42;
\end{align*}
\]

Thread B:
\[
\begin{align*}
\text{ans } &= i;
\end{align*}
\]

- What are the possible values for \texttt{ans}?

\[
\begin{align*}
i: \ &00000...00000111 \\
\vdots \\
i: \ &00000...00101010
\end{align*}
\]
Some actions are atomic

Thread A:
\[
\begin{align*}
\text{int } i &= 7; \\
i &= 42;
\end{align*}
\]

Thread B:
\[
\begin{align*}
\text{ans} &= i;
\end{align*}
\]

• What are the possible values for \texttt{ans}?

\[
\begin{align*}
i &: 00000...00000111 \\
\vdots \\
i &: 00000...00101010 \\
\text{ans} &: 00000...00101111
\end{align*}
\]

• In Java:
  • Reading an int variable is atomic
  • Writing an int variable is atomic

  • Thankfully, \texttt{ans}: 00000...00101111 is not possible
Bad news: some simple actions are not atomic

- Consider a single 64-bit `long` value

- Concurrently:
  - Thread 1 writing `sA` and `sB`
  - Thread 2 reading `sA` and `sB`

- What could Thread 2 see?
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.,
    
    ```java
    synchronized void foo() { ... } // locks "this"
    ```

    ```java
    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
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
    ```java
    int get();
    void set(int newValue);
    int getAndSet(int newValue);
    int getAndAdd(int delta);
    ...
    ```
  - AtomicIntegerArray
  - AtomicBoolean
  - AtomicLong
  - ...
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...
Next week:

- **Static analysis**
  - JSure: A static analysis tool for concurrent programs