Principles of Software Construction: Objects, Design, and Concurrency

The Perils of Concurrency, Part 3

*Can't live with it.*

*Can't live without it.*

**Charlie Garrod**  **Jonathan Aldrich**
Administrivia

- See Charlie if you still need your midterm exam
- Homework 5b due next Thursday, 11:59 p.m.
  - Finish by Friday (14 Nov) 10 a.m. if you want to be considered as a "Best Framework" for Homework 5c
    - Our evaluation considers:
      - Novelty
      - Functional correctness
      - Documentation
      - ...
- Homework 3 arena winners in class next week
Key concepts from Tuesday
Bad news: some simple actions are not atomic

- Consider a single 64-bit `long` value

  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
```

```
ans: 00000...00101010
```

```
ans: 01001...00101010
```

(10000000000)

(42)

(100000000042 or ...)
Key concepts from Tuesday

- Basic concurrency in Java
- Atomicity
- Race conditions
- The Java synchronized keyword
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
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.BlockingQueue

- Implements java.util.Queue\<E\>

- java.util.concurrent.ArrayBlockingQueue
  - put blocks if the queue is full
  - poll blocks if the queue is empty
  - Internally uses wait/notify

- java.util.concurrent.SynchronousQueue
  - Each put directly waits for a corresponding poll
  - Internally uses wait/notify
Today: Concurrency, part 3

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

- Basic concurrency in Java
  - Explicit synchronization with threads and shared memory
  - More concurrency problems

- Higher-level abstractions for concurrency
  - Data structures
  - Higher-level languages and frameworks
  - Hybrid approaches

- In the trenches of parallelism
  - Using the Java concurrency framework
  - Prefix-sums implementation
Concurreny at the language level

• Consider:
  ```java
  int sum = 0;
  Iterator i = coll.iterator();
  while (i.hasNext()) {
    sum += i.next();
  }
  ```

• In python:
  ```python
  sum = 0;
  for item in coll:
    sum += item
  ```
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
Parallel quicksort in Nesl

function quicksort(a) =
  if (#a < 2) then a
  else
    let pivot = a[#a/2];
    lesser = {e in a| e < pivot};
    equal = {e in a| e == pivot};
    greater = {e in a| e > pivot};
    result = {quicksort(v): v in [lesser,greater]};
    in result[0] ++ equal ++ result[1];

• Operations in {} occur in parallel

• What is the total work? What is the depth?
  • What assumptions do you have to make?
Prefix sums (a.k.a. inclusive scan)

- **Goal:** given array $x[0...n-1]$, compute array of the sum of each prefix of $x$

  \[
  \begin{array}{l}
  \text{sum}(x[0...0]), \\
  \text{sum}(x[0...1]), \\
  \text{sum}(x[0...2]), \\
  \vdots \\
  \text{sum}(x[0...n-1])
  \end{array}
  \]

- **e.g.,** $x = [13, 9, -4, 19, -6, 2, 6, 3]$
  
  prefix sums: $[13, 22, 18, 37, 31, 33, 39, 42]$
Parallel prefix sums

• Intuition: If we have already computed the partial sums $\text{sum}(x[0...3])$ and $\text{sum}(x[4...7])$, then we can easily compute $\text{sum}(x[0...7])$

• e.g., $x = [13, 9, -4, 19, -6, 2, 6, 3]$
Parallel prefix sums algorithm, winding

- Computes the partial sums in a more useful manner

\[
\begin{align*}
[13, & \quad 9, \quad -4, \quad 19, \quad -6, \quad 2, \quad 6, \quad 3] \\
[13, & \quad 22, \quad -4, \quad 15, \quad -6, \quad -4, \quad 6, \quad 9]
\end{align*}
\]
Parallel prefix sums algorithm, winding

- Computes the partial sums in a more useful manner

\[
\begin{align*}
[13, & \quad 9, \quad -4, \quad 19, \quad -6, \quad 2, \quad 6, \quad 3] \\
[13, & \quad 22, \quad -4, \quad 15, \quad -6, \quad -4, \quad 6, \quad 9] \\
[13, & \quad 22, \quad -4, \quad 37, \quad -6, \quad -4, \quad 6, \quad 5]
\end{align*}
\]
Parallel prefix sums algorithm, winding

- Computes the partial sums in a more useful manner

\[ [13, 9, -4, 19, -6, 2, 6, 3] \]
\[ [13, 22, -4, 15, -6, -4, 6, 9] \]
\[ [13, 22, -4, 37, -6, -4, 6, 5] \]
\[ [13, 22, -4, 37, -6, -4, 6, 42] \]
Parallel prefix sums algorithm, unwinding

• Now unwinds to calculate the other sums

\[ [13, 22, -4, 37, -6, -4, 6, 42] \]

\[ [13, 22, -4, 37, -6, 33, 6, 42] \]
Parallel prefix sums algorithm, unwinding

• Now unwinds to calculate the other sums

\[
\begin{array}{cccccccc}
13 & 22 & -4 & 37 & -6 & -4 & 6 & 42 \\
13 & 22 & -4 & 37 & -6 & 33 & 6 & 42 \\
13 & 22 & 18 & 37 & 31 & 33 & 39 & 42
\end{array}
\]

• Recall, we started with:

\[
\begin{array}{cccccccc}
13 & 9 & -4 & 19 & -6 & 2 & 6 & 3
\end{array}
\]
Parallel prefix sums

• Intuition: If we have already computed the partial sums \(\text{sum}(x[0\ldots3])\) and \(\text{sum}(x[4\ldots7])\), then we can easily compute \(\text{sum}(x[0\ldots7])\)

• e.g., \(x = [13, 9, -4, 19, -6, 2, 6, 3]\)

• Pseudocode:

```python
prefix_sums(x):
    for d in 0 to (\lg n)-1:  // d is depth
        parallel for i in 2^d-1 to n-1, by 2^{d+1}:
            x[i+2^d] = x[i] + x[i+2^d]

    for d in (\lg n)-1 to 0:
        parallel for i in 2^d-1 to n-1-2^d, by 2^{d+1}:
            if (i-2^d >= 0):
                x[i] = x[i] + x[i-2^d]
```
Parallel prefix sums algorithm, in code

- **An iterative Java-esque implementation:**

```java
void computePrefixSums(long[] a) {
    for (int gap = 1; gap < a.length; gap *= 2) {
        parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
            a[i+gap] = a[i] + a[i+gap];
        }
    }
    for (int gap = a.length/2; gap > 0; gap /= 2) {
        parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
            a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
        }
    }
}
```
Parallel prefix sums algorithm, in code

- A recursive Java-esque implementation:
  ```java
  void computePrefixSumsRecursive(long[] a, int gap) {
    if (2*gap - 1 >= a.length) {
      return;
    }

    parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
      a[i+gap] = a[i] + a[i+gap];
    }

    computePrefixSumsRecursive(a, gap*2);

    parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
      a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
    }
  }
  ```
Parallel prefix sums algorithm

• How good is this?
Parallel prefix sums algorithm

- How good is this?
  - Work: $O(n)$
  - Depth: $O(\lg n)$

- See Main.java, PrefixSumsNonconcurrentParallelWorkImpl.java
Goal: parallelize the PrefixSums implementation

- Specifically, parallelize the parallelizable loops
  ```c
  parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
    a[i+gap] = a[i] + a[i+gap];
  }
  ```

- Partition into multiple segments, run in different threads
  ```c
  for(int i=left+gap-1; i+gap<right; i += 2*gap) {
    a[i+gap] = a[i] + a[i+gap];
  }
  ```
Recall the Java primitive concurrency tools

- **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();
  ```
Recall the Java primitive concurrency tools

- The `java.langRunnable` interface
  ```java
define
  public interface Runnable {
      void run();
  }
```

- The `java.langThread` class
  ```java
  public class Thread {
      public Thread(Runnable r);
      public void start();
      public static void sleep(long millis);
      public void join();
      public boolean isAlive();
      public static Thread currentThread();
  }
```

- The `java.util.concurrentCallable<V>` interface
  - Like `java.langRunnable` but can return a value
  ```java
  public interface Callable<V> {
      public V call();
  }
```
A framework for asynchronous computation

- The `java.util.concurrent.Future<V> interface`
  
  ```java
  V get();
  V get(long timeout, TimeUnit unit);
  boolean isDone();
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  ```
A framework for asynchronous computation

- **The java.util.concurrent.Future\<V> interface**
  
  ```java
  V get();
  V get(long timeout, TimeUnit unit);
  boolean isDone();
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  ```

- **The java.util.concurrent.ExecutorService interface**
  
  ```java
  Future submit(Runnable task);
  Future\<V> submit(Callable\<V> task);
  List\<Future\<V>> invokeAll(Collection\<Callable\<V>> tasks);
  Future\<V> invokeAny(Collection\<Callable\<V>> tasks);
  ```
Executors for common computational patterns

- From the `java.util.concurrent.Executors` class:
  - `static ExecutorService newSingleThreadExecutor();`
  - `static ExecutorService newFixedThreadPool(int n);`
  - `static ExecutorService newCachedThreadPool();`
  - `static ExecutorService newScheduledThreadPool(int n);`

- Aside: see `NetworkServer.java` (later)
Fork/Join: another common computational pattern

- **In a long computation:**
  - Fork a thread (or more) to do some work
  - Join the thread(s) to obtain the result of the work
Fork/Join: another common computational pattern

• **In a long computation:**
  - Fork a thread (or more) to do some work
  - Join the thread(s) to obtain the result of the work

• **The `java.util.concurrent.ForkJoinPool` class**
  - Implements `ExecutorService`
  - Executes `java.util.concurrent.ForkJoinTask<V>` or `java.util.concurrent.RecursiveTask<V>` or `java.util.concurrent.RecursiveAction`
The RecursiveAction abstract class

```java
public class MyActionFoo extends RecursiveAction {
    public MyActionFoo(...) {
        store the data fields we need
    }

    @Override
    public void compute() {
        if (the task is small) {
            do the work here;
            return;
        }

        invokeAll(new MyActionFoo(...), // smaller
                  new MyActionFoo(...), // tasks
                  ...);
    }
}
```
A ForkJoin example

- See PrefixSumsParallelImpl.java, PrefixSumsParallelLoop1.java, and PrefixSumsParallelLoop2.java
- See the processor go, go go!
Parallel prefix sums algorithm

• How good is this?
  ▪ Work: \( O(n) \)
  ▪ Depth: \( O(\lg n) \)

• See PrefixSumsSequentialImpl.java
Parallel prefix sums algorithm

• How good is this?
  ▪ Work: $O(n)$
  ▪ Depth: $O(\lg n)$

• See PrefixSumsSequentialImpl.java
  ▪ $n-1$ additions
  ▪ Memory access is sequential

• For PrefixSumsNonsequentialImpl.java
  ▪ About $2n$ useful additions, plus extra additions for the loop indexes
  ▪ Memory access is non-sequential

• The punchline: Constants matter.
Next week…

• Introduction to distributed systems
In-class example for parallel prefix sums

\[ [ 7, \ 5, \ 8, \ -36, \ 17, \ 2, \ 21, \ 18 ] \]