# Principles of Software Construction: Objects, Design and Concurrency

## The Perils of Concurrency, Part 3

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

***15-214***

***toad***

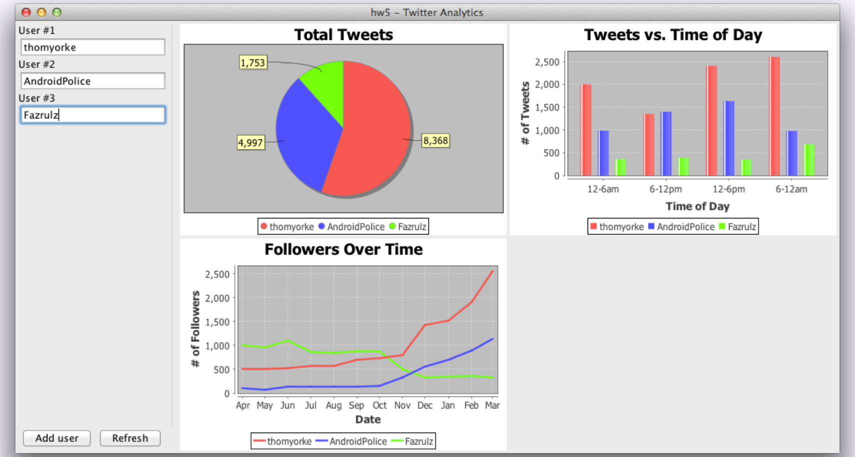
Fall 2013

Jonathan Aldrich **Charlie Garrod**

School of Computer Science
institute for SOFTWARE RESEARCH

# Administrivia

- Homework 5: The Framework Strikes Back
  - 5a presentations tomorrow!
    - We will re-publish room assignments via Piazza
    - Commit/push design & presentation by 8:59 a.m.



# Key topics from last Thursday

# Dealing with deadlock

- 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

# Concurrency control in Java

- Using primitive synchronization, you are responsible for correctness:
  - Avoiding race conditions
  - Progress (avoiding deadlock and livelock)
- Java provides tools to help:
  - `volatile` fields
  - `java.util.concurrent.atomic`
  - `java.util.concurrent`

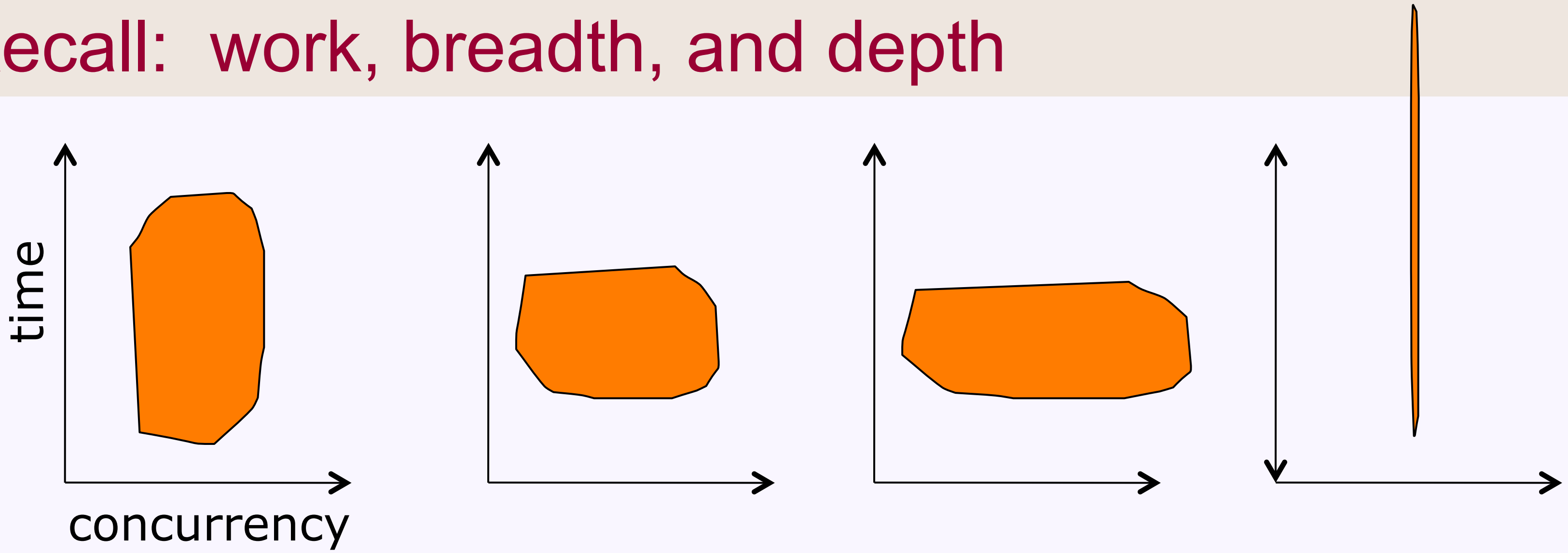
# Aside: The power of immutability

- Recall: Data is *mutable* if it can change over time. Otherwise it is *immutable*.
  - Primitive data declared as `final` is always immutable
- After immutable data is initialized, it is immune from race conditions

# Today: More concurrency

- High-level abstractions of concurrency
- In the trenches of parallelism
  - Using the Java concurrency framework
  - Prefix-sums implementation

# Recall: work, breadth, and depth




- Work: total effort required
  - area of the shape
- Breadth: extent of simultaneous activity
  - width of the shape
- Depth (or span): length of longest computation
  - height of the shape

# Concurrency at the language level

- Consider:

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

- In python:

```
sum = 0;
for item in coll:
     sum += item
```

# 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`

```
[ sum(x[0…0]),
  sum(x[0…1]),
  sum(x[0…2]),
  …
  sum(x[0…n-1]) ]
```

- 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 `sum(x[0…3])` and `sum(x[4…7])`, then we can easily compute `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

```
[13,   9,  -4,  19,  -6,   2,   6,   3]

[13,  22,  -4,  15,  -6,  -4,   6,   9]
```

# 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]
```

# 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

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

[13,   22,   -4,   37,   -6,   33,    6,   42]

[13,   22,   18,   37,   31,   33,   39,   42]
```

- Recall, we started with:

```
[13,    9,   -4,   19,   -6,    2,    6,    3]
```

# Parallel prefix sums

- Intuition: If we have already computed the partial sums `sum(x[0…3])` and `sum(x[4…7])`, then we can easily compute `sum(x[0…7])`
- e.g., `x = [13, 9, -4, 19, -6, 2, 6, 3]`
- Pseudocode:

```
prefix_sums(x):
  for d in 0 to (lg n)-1:          // d is depth
    parallelfor 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:
    parallelfor 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:

```
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:

```
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, PrefixSumsNonSequentialImpl.java

# Goal: parallelize PrefixSumsNonSequentialImpl

- Specifically, parallelize the parallelizable loops

```
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

```
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

```
void              run();
```

- The `java.lang.Thread` class

```
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.lang.Runnable` interface

```
void            run();
```

- The `java.lang.Thread` class

```
Thread(Runnable r);
void            start();
static void     sleep(long millis);
void            join();
boolean         isAlive();
static Thread currentThread();
```

- The `java.util.concurrent.Callable<V>` interface
  - Like `java.lang.Runnable` but can return a value

```
V               call();
```

# A framework for asynchronous computation

- The `java.util.concurrent.Future<V>` interface

```
V       get();
V       get(long timeout, TimeUnit unit);
boolean isDone();
boolean cancel(boolean mayInterruptIfRunning);
boolean isCancelled();
```

- The `java.util.concurrent.ExecutorService` interface

```
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

```
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 time...