SIGCSE Workshop 19, March 11, 2011
Multithreading Pretty Early for Everyone

Exercises

Parallelism and The Fork-Join Framework

I recommend starting with the first two exercises. You are unlikely to have time to code up more during the workshop, but you could skip ahead or sketch the ideas for the more challenging exercises (next page).

1. Use a parallel *reduction* to compute the *leftmost* `String` *that starts with an* `'S'` in a `String[]` (or `null` if there is none).

Details: In `WorkshopReductions.java` complete the `class LeftmostStartsWith` definition that starts on line 76. The code template immediately above this in the file will be extremely useful. Running `ReductionUtils.main` will test your code; a correct solution will produce:

```
Testing Reductions

Test LeftmostStartsWith
test1 SUCHO
test2 S
test3 null
test4 Salamander
test5 Swan
```

2. Use a parallel *map* to update a `String[]` such that *every entry containing a string starting with* `'S'` *is replaced with* `"[redacted]"`.

Details: In `WorkshopMaps.java` complete the `class RedactSWords` definition that starts on line 74. The code template immediately above this in the file will be extremely useful. Running `MapUtils.main` will test your code; a correct solution will produce:

```
Testing Maps

Test RedactSWords
test1 redacted 38 words
test2 redacted 3865 words
test3 redacted 0 words
test4 redacted 1 words
test5 redacted 2 words
```

The remaining ForkJoin exercises are similar, with 3–5 using `WorkshopReductions.java` and 6 using `WorkshopMaps.java`. To test your solutions, uncomment appropriate lines in `ReductionUtils.main` or `MapUtils.main`. The correct test output for all exercises is posted at:
`www.cs.washington.edu/homes/djg/teachingMaterials/grossmanSPAC_sigcse_workshop_2011/reductionResults.txt`
`www.cs.washington.edu/homes/djg/teachingMaterials/grossmanSPAC_sigcse_workshop_2011/mapResults.txt`

3. Use a parallel *reduction* to compute the *index of the leftmost* `String` *that starts with an* `'S'` in a `String[]` (or `-1` if there is none). (Use `class LeftmostStartsWithIndex`.)

4. Use a parallel *reduction* to compute the *second leftmost* `String` *that starts with an* `'S'` in a `String[]` (or `null` if there is none). (Use `class SecondLeftmost`.)

5. Use a parallel *reduction* to compute the k^{th} *leftmost* `String` *that starts with an* `'S'` in a `String[]` (or `null` if there is none). k is passed as a parameter and you can assume it is fairly small. (Use `class KthLeftmost`.)

6. Use a parallel *map* to write a *generic* string-replacement algorithm: Take in a `Changer` object (defined in `WorkshopMaps.java`) and apply its `m` method to every element of a `String[]`, replacing each `String` with the result of the call to `m`. (Use `class Redact`.)

Concurrency

Consider this code skeleton (as on the slides), which has no data races.

```
class Stack<E> {
  ... // private state used by isEmpty, push, pop
  synchronized boolean isEmpty() { ... }
  synchronized void push(E val) { ... }
  synchronized E pop() {
     if(isEmpty())
       throw new StackEmptyException();
     ...
  }
  E peek() { // this is wrong
     E ans = pop();
     push(ans);
     return ans;
  }
}
```

1. Show an interleaving where:

- Thread 1 executes (the statements in) `peek`
- Thread 2 does two pushes then a pop
- The result of the pop is not the most recently pushed value

2. Show an interleaving where:

- Two threads both execute (the statements in) `peek`
- One thread throws an exception indicating the stack is empty
- (Also describe the state of the stack before these operations that leads to an exception.)

3. (More Challenging?) Argue that N threads executing simultaneous `peek` operations could leave the N shallowest elements on the stack in *any order*.

4. (Unrelated to stacks) Why is this code skeleton ***wrong***? Assume `f` and `g` are called by different threads.

```
class C {
  boolean stop = false;
  boolean done = false;
  Result bestSoFar = ... ;
  void f() {
    while(!stop) {
      ... // keep iteratively improving a result
      bestSoFar = ... ;
    }
    done = true;
  }
  Result g() {
    while(!didUserAskToStop()) { /* spin */ }
    stop = true;
    while(!done) { /* spin */ }
    return bestSoFar;
  }
}
```