Principles of Software Construction: Objects, Design, and Concurrency

Part 42: Concurrency

Introduction to concurrency frameworks

In the trenches of parallelism

Josh Bloch    Charlie Garrod    Darya Melicher
Administrivia

- Homework 5b due 11:59 p.m. tonight
  - Turn in by Wednesday 9 a.m. to be considered as a Best Framework
- Optional reading due today:
  - Java Concurrency in Practice, Chapter 12
Key concepts from last Thursday
Strategies for thread safety

- **Thread-confined state** – mutate but don’t share
- **Shared read-only state** – share but don’t mutate
- **Shared thread-safe** – object synchronizes itself internally
- **Shared guarded** – client synchronizes object(s) externally
Advice for building thread-safe objects

• Do as little as possible in synchronized region: get in, get out
  – Obtain lock
  – Examine shared data
  – Transform as necessary
  – Drop the lock

• If you must do something slow, move it outside the synchronized region
Shared thread-safe

- Thread-safe objects that perform internal synchronization
- You can build your own, but...
- You’re better off using ones from java.util.concurrent
Concurrent observer pattern with a subtle bug
This code is prone to liveness and safety failures!

```java
private final List<Observer<E>> observers = new ArrayList<>();
public void addObserver(Observer<E> observer) {
    synchronized(observers) { observers.add(observer); } 
}
public boolean removeObserver(Observer<E> observer) {
    synchronized(observers) { return observers.remove(observer); } 
}
private void notifyOf(E element) {
    synchronized(observers) {
        for (Observer<E> observer : observers) {
            observer.notify(this, element);
        }
    }
}
```
Concurrent observer pattern with a subtle bug
This code is prone to liveness and safety failures!

private final List<Observer<E>> observers = new ArrayList<>();
public void addObserver(Observer<E> observer) {
    synchronized(observers) { observers.add(observer); }
}
public boolean removeObserver(Observer<E> observer) {
    synchronized(observers) { return observers.remove(observer); }
}
private void notifyOf(E element) {
    synchronized(observers) {
        for (Observer<E> observer : observers)
            observer.notify(this, element);  // Callback while
               // holding lock!
    }
}
One solution: *snapshot iteration*

```java
private void notifyOf(E element) {
    List<Observer<E>> snapshot = null;

    synchronized( observers ) {
        snapshot = new ArrayList<>( observers );
    }

    for ( Observer<E> observer : snapshot ) {
        observer.notify( this, element ); // Safe
    }
}
```
A better solution: CopyOnWriteArrayList

private final List<Observer<E>> observers =
    new CopyOnWriteArrayList<>();

public void addObserver(Observer<E> observer) {
    observers.add(observer);
}

public boolean removeObserver(Observer<E> observer) {
    return observers.remove(observer);
}

private void notifyOf(E element) {
    for (Observer<E> observer : observers)
        observer.notify(this, element);
}
Today

- Concurrency in practice: In the trenches of parallelism
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class LittleTest {
    int number;

    @Test
    public void test() throws InterruptedException {
        number = 0;
        Thread t = new Thread(() -> {
            assertEquals(2, number);
        });
        number = 1;
        t.start();
        number++;
        t.join();
    }
}
How often does this test pass?

import org.junit.Test;
import static org.junit.Assert.assertEquals;

class LittleTest {  
    int number;

    @Test
    public void test() throws InterruptedException {
        number = 0;
        Thread t = new Thread(() -> {
            assertEquals(2, number);
        });
        number = 1;
        t.start();
        number++;
        t.join();
    }
}
How often does this test pass?

(a) It always fails
(b) It sometimes passes
(c) It always passes – but it tells us nothing
(d) It always hangs

JUnit doesn’t see assertion failures in other threads
import org.junit.*;
import static org.junit.Assert.*;

class LittleTest {
    int number;

    @Test
    public void test() throws InterruptedException {
        number = 0;
        Thread t = new Thread(() -> {
            assertEquals(2, number); // JUnit never sees the exception!
        });
        number = 1;
        t.start();
        number++;
        t.join();
    }
}
How do you fix it? (1)

// Keep track of assertion failures during test
volatile Exception exception;
volatile Error error;

// Triggers test case failure if any thread asserts failed
@After
public void tearDown() throws Exception {
    if (error != null)
        throw error;
    if (exception != null)
        throw exception;
}
How do you fix it? (2)

```
Thread t = new Thread(() -> {
    try {
        assertEquals(2, number);
    } catch (Error e) {
        error = e;
    } catch (Exception e) {
        exception = e;
    }
});
```

*YMMV (It’s a race condition)

*Now it sometimes passes*
The moral

- JUnit does not well-support concurrent tests
  - You might get a false sense of security
- Concurrent clients beware...
Puzzler: “Ping Pong”

public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread(() -> pong());
        t.run();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
What does it print?

public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread(() -> pong());
        t.run();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}

(a) PingPong
(b) PongPing
(c) It varies
(d) None of the above
What does it print?

(a) PingPong  
(b) PongPing  
(c) It varies  
(d) None of the above

Not a multithreaded program!
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread(() -> pong());
        t.run(); // An easy typo!
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}
public class PingPong {
    public static synchronized void main(String[] a) {
        Thread t = new Thread(() -> pong());
        t.start();
        System.out.print("Ping");
    }

    private static synchronized void pong() {
        System.out.print("Pong");
    }
}

Now prints PingPong
The moral

• Invoke `Thread.start`, not `Thread.run`
• `java.lang.Thread` should not have implemented `Runnable`
Today

• Concurrency in practice: In the trenches of parallelism
Concurrency at the language level

• Consider:
  Collection<Integer> collection = ...;
  int sum = 0;
  for (int i : collection) {
    sum += i;
  }

• In python:
  collection = ...
  sum = 0
  for item in collection:
    sum += item
Parallel quicksort in Nesl

```haskell
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
• 210-esque questions: What is total work? What is depth?
Prefix sums (a.k.a. inclusive scan, a.k.a. scan)

- Goal: given array $x[0...n-1]$, compute array of the sum of each prefix of $x$
  
  $[ \text{sum}(x[0...0]), \text{sum}(x[0...1]), \text{sum}(x[0...2]), ... \text{sum}(x[0...n-1]) ]$

- 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 \( \text{sum}(x[0...3]) \) 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, *upsweep*

Compute the partial sums in a more useful manner

\[
\begin{bmatrix}
13, & 9, & -4, & 19, & -6, & 2, & 6, & 3 \\
13, & 22, & -4, & 15, & -6, & -4, & 6, & 9
\end{bmatrix}
\]
Parallel prefix sums algorithm, upsweep

Compute the partial sums in a more useful manner

\[
\begin{bmatrix}
13, & 9, & -4, & 19, & -6, & 2, & 6, & 3 \\
13, & 22, & -4, & 15, & -6, & -4, & 6, & 9 \\
13, & 22, & -4, & 37, & -6, & -4, & 6, & 5
\end{bmatrix}
\]
Parallel prefix sums algorithm, \textit{upsweep}

Compute the partial sums in a more useful manner

\[
\begin{bmatrix}
13, & 9, & -4, & 19, & -6, & 2, & 6, & 3 \\
13, & 22, & -4, & 19, & -6, & -4, & 6, & 9 \\
13, & 22, & -4, & 37, & -6, & -4, & 6, & 5 \\
13, & 22, & -4, & 37, & -6, & -4, & 6, & 42 \\
\end{bmatrix}
\]
Parallel prefix sums algorithm, \textit{downsweep}

Now unwind to calculate the other sums

\begin{align*}
[13, & \quad 22, \quad -4, \quad 37, \quad -6, \quad -4, \quad 6, \quad 42] \\
[13, & \quad 22, \quad -4, \quad 37, \quad -6, \quad 33, \quad 6, \quad 42]
\end{align*}
Parallel prefix sums algorithm, **downsweep**

Now unwind to calculate the other sums

\[
\begin{bmatrix}
13, & 22, & -4, & 37, & -6, & -4, & 6, & 42 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
13, & 22, & -4, & 37, & -6, & 33, & 6, & 42 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
13, & 22, & 18, & 37, & 31, & 33, & 39, & 42 \\
\end{bmatrix}
\]

• Recall, we started with:

\[
\begin{bmatrix}
13, & 9, & -4, & 19, & -6, & 2, & 6, & 3 \\
\end{bmatrix}
\]
Doubling array size adds two more levels

Upsweep

Downsweep
Parallel prefix sums

*pseudocode*

```
// Upsweep
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]

// Downsweep
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 iterativePrefixSums(long[] a) {
    int gap = 1;
    for (; 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 (; gap > 0; gap /= 2) {
        parfor(int i=gap-1; i < 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 recursivePrefixSums(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];
    }

    recursivePrefixSums(a, gap*2);

    parfor(int i=gap-1; i < 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 PrefixSums.java,
  PrefixSumsSequentialWithParallelWork.java
Goal: parallelize the PrefixSums implementation

- Specifically, parallelize the parallelizable loops
  
  ```java
  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
  
  ```java
  for(int i = left+gap-1; i+gap < right; i += 2*gap) {
    a[i+gap] = a[i] + a[i+gap];
  }
  ```
Recall from Thursday: Fork/join in Java

• 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`

• 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 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(...), // subtasks
                  ...);                // ...
    }
}
A ForkJoin example

- See PrefixSumsParallelForkJoin.java
- See the processor go, go go!
Parallel prefix sums algorithm

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

• See `PrefixSumsParallelArrays.java`
Parallel prefix sums algorithm

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

• See PrefixSumsParallelArrays.java
• See PrefixSumsSequential.java
Parallel prefix sums algorithm

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

- See PrefixSumsParallelArrays.java
- See PrefixSumsSequential.java
  - $n-1$ additions
  - Memory access is sequential

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

- The punchline:
  - Don't roll your own
  - Cache and constants matter
In-class example for parallel prefix sums

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