Principles of Software Construction: Objects, Design, and Concurrency

The Perils of Concurrency, part 3

Can't live with it.
Cant live without it. No joke.

Spring 2014

Charlie Garrod   Christian Kästner

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 5b due Tuesday night
  - Turn in by Thursday, 10 April, 10:00 a.m. to be considered as framework-supporting team
- Homework 2 Arena winners in class Tuesday
- Looking for summer research opportunities?
  - http://www.isri.cmu.edu/education/reu-se/index.html

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

Key concepts from Tuesday

- Basic concurrency in Java
  - `java.lang.Runnable`
  - `java.lang.Thread`
- Atomicity
- Race conditions
- The Java `synchronized` keyword

Basic concurrency in Java

- 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();
```

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

Primitive concurrency control in Java

- `java.lang.Object` allows some coordination via the intrinsic lock:

```
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

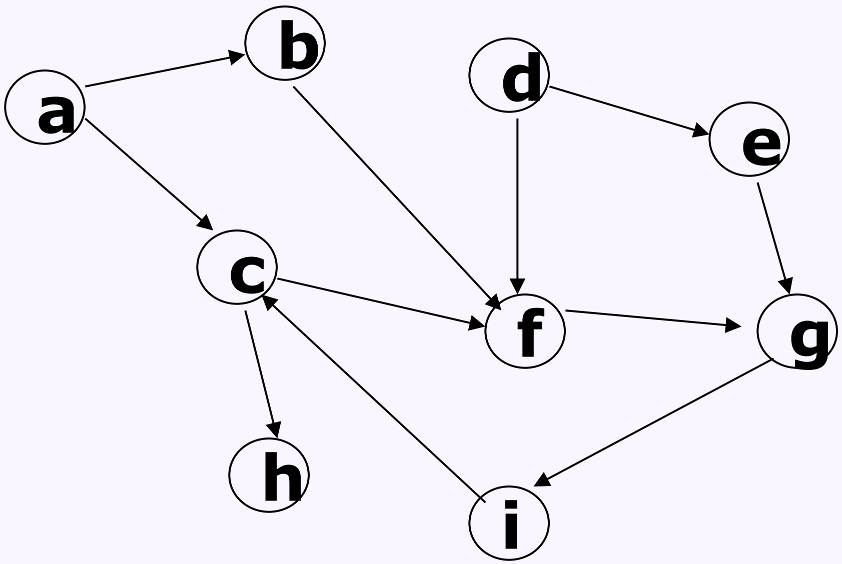
```
synchronized(lock1) {
    // do stuff that requires lock1

    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

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



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…
```

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

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

Concurrency 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`
  - Java concurrency framework

The `java.util.concurrent.atomic` package

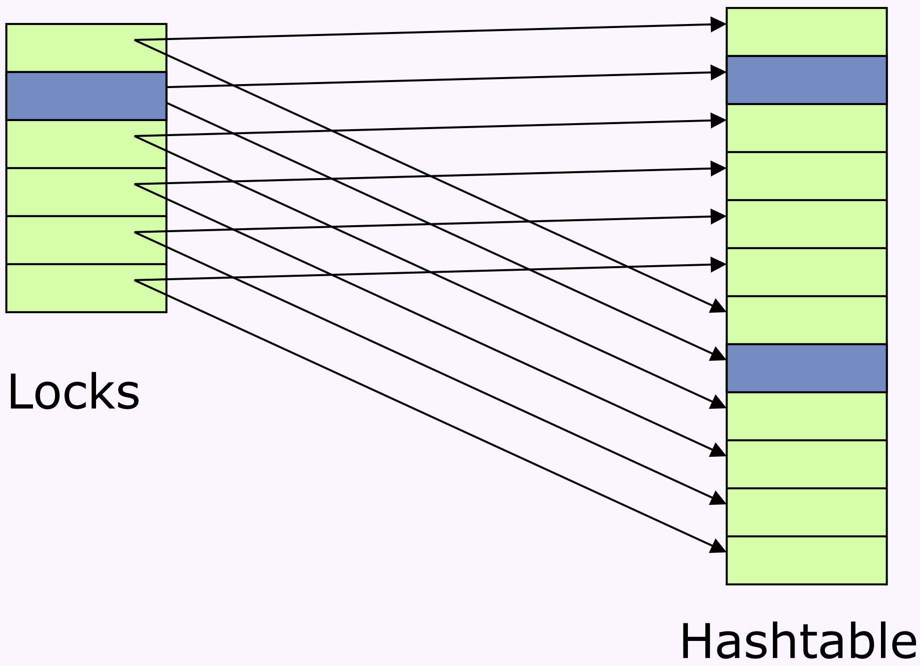
- Concrete classes supporting atomic operations
  - `AtomicInteger`
    ```
    int  get();
    void set(int newValue);
    int  getAndSet(int newValue);
    int  getAndAdd(int delta);
    …
    ```
  - `AtomicIntegerArray`
  - `AtomicBoolean`
  - `AtomicLong`
  - …

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…



java.util.concurrent.BlockingQueue

- Implements java.util.Queue<E>
- java.util.concurrent.SynchronousQueue
  - Each put directly waits for a corresponding poll
  - Internally uses wait/notify
- java.util.concurrent.ArrayBlockingQueue
  - put blocks if the queue is full
  - poll blocks if the queue is empty
  - Internally uses wait/notify

The `CopyOnWriteArrayList`

- Implements `java.util.List<E>`
- All writes to the list copy the array storing the list elements

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

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
```

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`

```
[ 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

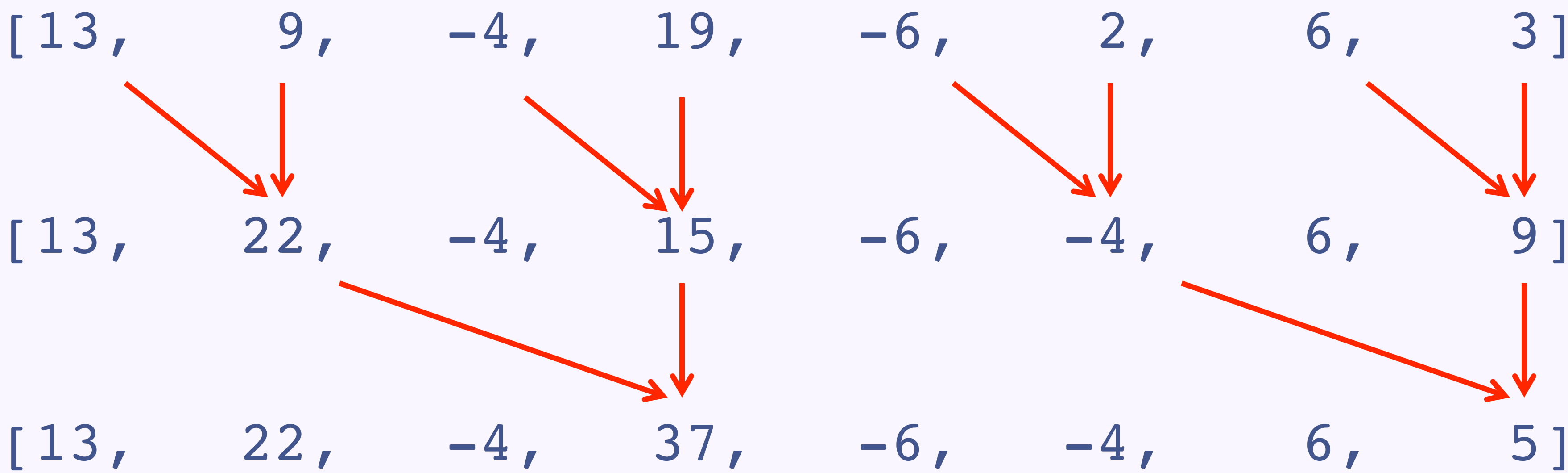
- Computes the partial sums in a more useful manner

```
[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



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, PrefixSumsNonconcurrentParallelWorkImpl.java

Goal: parallelize the PrefixSums implementation

- 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…