Objects Analysis Threads Design 15-214

Principles of Software Construction: Objects, Design, and Concurrency

The Perils of Concurrency, Part 2

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

Fall 2014

Charlie Garrod Jonathan Aldrich

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Midterm exam returned at end of class today
- Homework 5a due 8:59 a.m. tomorrow morning
- Do you want to be a Software Engineer?

The foundations of the Software Engineering minor

- Core computer science fundamentals
- Building good software
- Organizing a software project
  - Development teams, customers, and users
  - Process, requirements, estimation, management, and methods
- The larger context of software
  - Business, society, policy
- Engineering experience
- Communication skills
  - Written and oral

institute for SOFTWARE RESEARCH

SE minor requirements

- Prerequisite: 15-214
- Two core courses
  - 15-313 Foundations of SE (fall semesters)
  - 15-413 SE Practicum (spring semesters)
- Three electives
  - Technical
  - Engineering
  - Business or policy
- Software engineering internship + reflection
  - 8+ weeks in an industrial setting, then
  - 17-413

To apply to be a Software Engineering minor

- Email aldrich@cs.cmu.edu and clegoues@cs.cmu.edu
  - Your name, Andrew ID, class year, QPA, and minor/majors
  - Why you want to be a SE minor
  - Proposed schedule of coursework

- Spring applications due by Friday, 7 Nov 2014
  - Only 15 SE minors accepted per graduating class

- More information at:
  - http://isri.cmu.edu/education/undergrad/

Key concepts from last Tuesday

Power requirements of a CPU

- Approx.: **C**apacitance * **V**oltage^2 * **F**requency
- To increase performance:
  - More transistors, thinner wires: more **C**
    - More power leakage: increase **V**
  - Increase clock frequency **F**
    - Change electrical state faster: increase **V**
- Problem: Power requirements are super-linear to performance
  - Heat output is proportional to power input

Problems of concurrency

- Realizing the potential
  - Keeping all threads busy doing useful work
- Delivering the right language abstractions
  - How do programmers think about concurrency?
  - Aside: parallelism vs. concurrency
- Non-determinism
  - Repeating the same input can yield different results

Atomicity

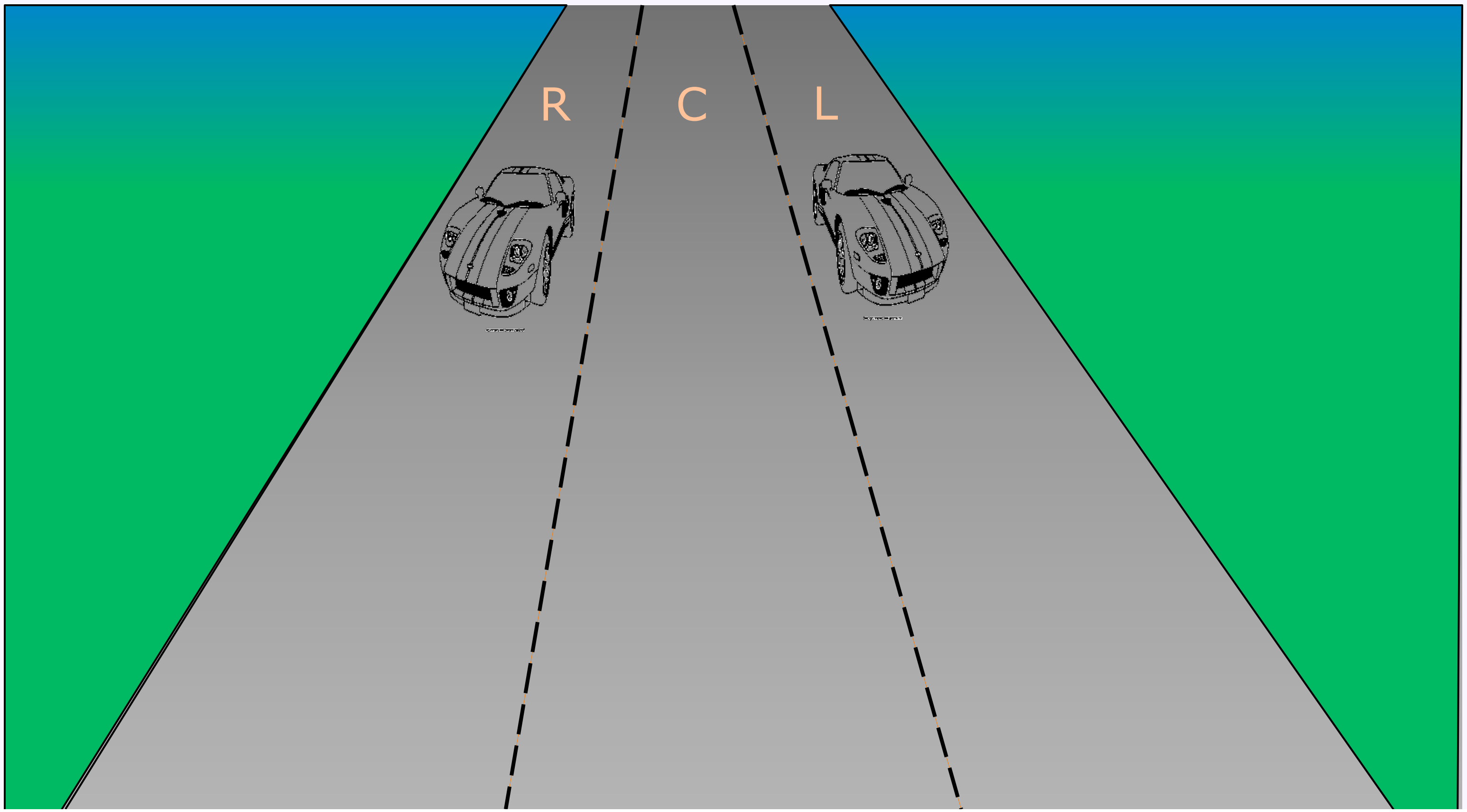
- An action is *atomic* if it is indivisible
  - Effectively, it happens all at once
    - No effects of the action are visible until it is complete
    - No other actions have an effect during the action
- In Java, integer increment is not atomic

`i++;` is actually

1. Load data from variable `i`
2. Increment data by `1`
3. Store data to variable `i`

Race conditions in real life

- E.g., check-then-act on the highway



Race conditions in *your* real life

- E.g., check-then-act in simple code

```
public class StringConverter {
    private Object o;
    public void set(Object o) {
        this.o = o;
    }
    public String get() {
        if (o == null) return "null";
        return o.toString();
    }
}
```

  - See StringConverter.java, Getter.java, Setter.java

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

`i:` **00000…00000111**

⋮

`i:` **00000…00101010**

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

`i:` **00000…00000111**

⋮

`i:` **00000…00101010**

- In Java:
  - Reading an int variable is atomic
  - Writing an int variable is atomic
  - Thankfully, `ans:` **00000…00101111** is not possible

Bad news: some simple actions are not atomic

- Consider a single 64-bit `long` value

high bits	low bits

  - 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** (10000000000)

`ans:` **00000…00101010** (42)

`ans:` **01001…00101010** (10000000042 or …)

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

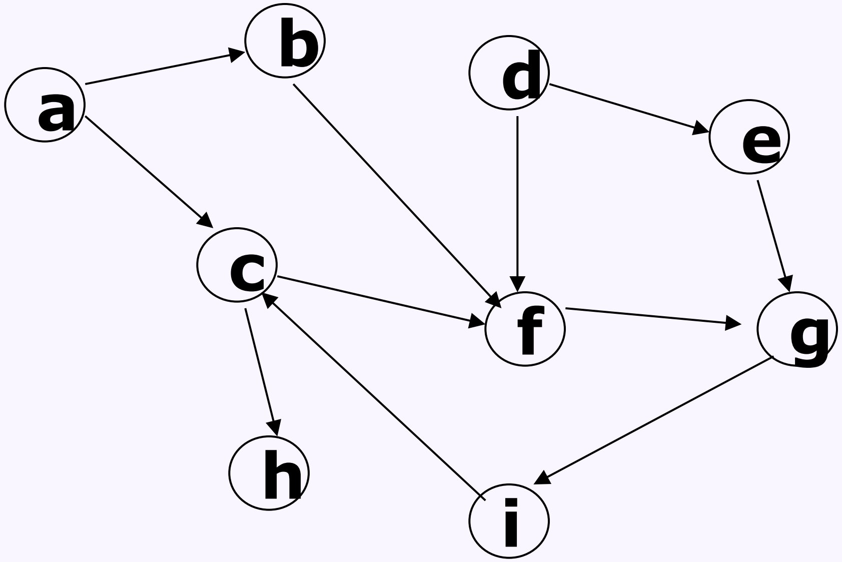
- E.g., Alice and Bob, unaware of each other, both need file *A* and network connection *B*
  - Alice gets lock for file *A*
  - Bob gets lock for network connection *B*
  - Alice tries to get lock for network connection *B*, and waits…
  - Bob tries to get lock for file *A*, and waits…
- See Counter.java and DeadlockExample.java

Dealing with deadlock (abstractly, not with Java)

- Detect deadlock
  - Statically?
  - Dynamically at run time?
- Avoid deadlock
- Alternative approaches
  - Automatic restarts
  - Optimistic concurrency control

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

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
- 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

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`

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

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

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);
boolean compareAndSet(int expectedValue,
                      int newValue);

…
```

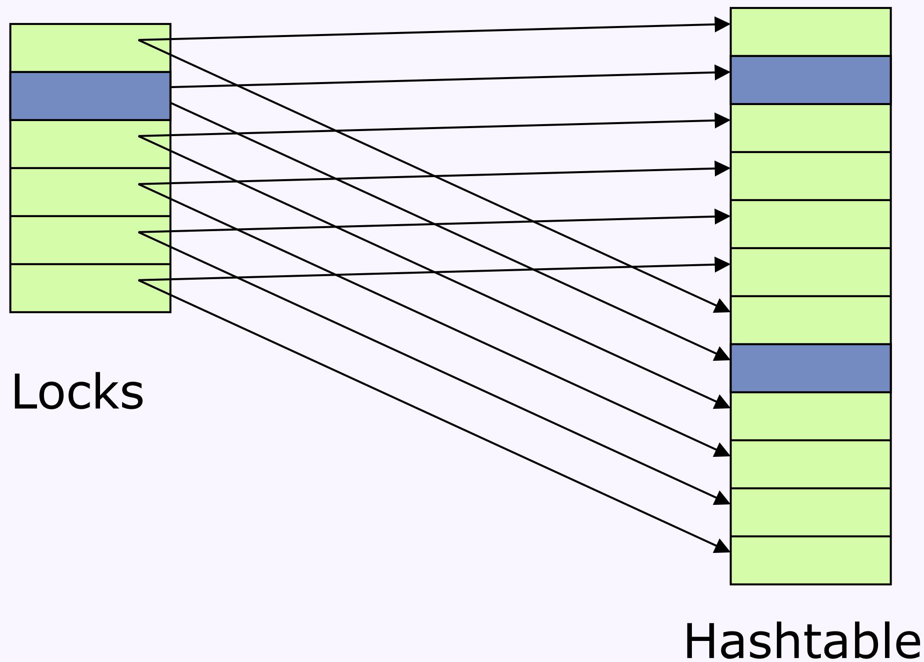
  - `AtomicIntegerArray`
  - `AtomicBoolean`
  - `AtomicLong`
  - …

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