Principles of Software Construction: Objects, Design, and Concurrency

The Perils of Concurrency, part 2
Can't live with it.
Can't live without it.

Charlie Garrod  Christian Kästner
Administrivia

• Midterm exam returned at end of class today

• Homework 5a due tomorrow, 8:59 a.m.
  • 5b due the next Tuesday (08 April)
  • Turn in by Thursday, 10 April, 10:00 a.m. to be considered as framework-supporting team
  • 5c due the following Tuesday (15 April)

• 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
SE minor requirements

• Prerequisite: 15-214

• Two core courses
  - 15-313 (fall semesters)
  - 15-413 (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 jonathan.aldrich@cs.cmu.edu and poprocky@cs.cmu.edu
  ▪ Your name, Andrew ID, class year, QPA, and minor/majors
  ▪ Why you want to be a software engineer
  ▪ Proposed schedule of coursework

• Spring applications due by Friday, 11 Apr 2014
  ▪ Only 15 SE minors accepted per graduating class

• More information at:
  ▪ http://isri.cmu.edu/education/undergrad/
Key concepts from last Tuesday
Realizing the potential

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

- **Typical goals in parallel algorithm design?**
  - First minimize depth (total time we wait), then minimize work...
Today: Concurrency, part 2

- **The backstory**
  - Motivation, goals, problems, ...

- **Basic concurrency in Java**
  - Explicit synchronization with threads and shared memory
  - More concurrency problems

- **Coming soon:**
  - Higher-level abstractions for concurrency
    - Data structures
    - Higher-level languages and frameworks
    - Hybrid approaches
Amdahl’s law: How good can the depth get?

• **Ideal parallelism with** $N$ **processors:**
  - Speedup = $N$

• **In reality, some work is always inherently sequential**
  - Let $F$ be the portion of the total task time that is inherently sequential
  - Speedup = $\frac{1}{F + (1 - F)/N}$

  - Suppose $F = 10\%$. What is the max speedup? (you choose $N$)
    - As $N$ approaches $\infty$, $1/(0.1 + 0.9/N)$ approaches 10.
Using Amdahl’s law as a design guide

• For a given algorithm, suppose
  - $N$ processors
  - Problem size $M$
  - Sequential portion $F$

• An obvious question:
  - What happens to speedup as $N$ scales?

• Another important question:
  - What happens to $F$ as problem size $M$ scales?

"For the past 30 years, computer performance has been driven by Moore’s Law; from now on, it will be driven by Amdahl’s Law."

— Doron Rajwan, Intel Corp
Abstractions of concurrency

- **Processes**
  - Execution environment is isolated
    - Processor, in-memory state, files, ...
  - Inter-process communication typically slow, via message passing
    - Sockets, pipes, ...

- **Threads**
  - Execution environment is shared
  - Inter-thread communication typically fast, via shared state
Aside: Abstractions of concurrency

• What you see:
  ▪ State is all shared

• A (slightly) more accurate view of the hardware:
  ▪ Separate state stored in registers and caches
  ▪ Shared state stored in caches and memory
Basic concurrency in Java

• The `java.lang.Runnable` interface
  
  ```java
  void run();
  ```

• The `java.lang.Thread` class
  
  ```java
  Thread(Runnable r);
  void start();
  static void sleep(long millis);
  void join();
  boolean isAlive();
  static Thread currentThread();
  ```

• See IncrementTest.java
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

```java
i++;
```

is actually

1. Load data from variable `i`
2. Increment data by 1
3. Store data to variable `i`
One concurrency problem: race conditions

- A *race condition* is when multiple threads access shared data and unexpected results occur depending on the order of their actions.

- E.g., from IncrementTest.java:
  - Suppose `classData` starts with the value 41:

Thread A:
```
classData++;  // 1A. Load data(41) from classData
```

Thread B:
```
classData++;  // 1B. Load data(41) from classData
```

One possible interleaving of actions:
```
2A. Increment data(41) by 1 -> 42
2B. Increment data(41) by 1 -> 42
3A. Store data(42) to classData
3B. Store data(42) to classData
```

```
Race conditions in real life

- E.g., check-then-act on the highway
Race conditions in real life

- E.g., check-then-act at the bank
  - The "debit-credit problem"

### Alice, Bob, Bill, and the Bank

#### A. Alice to pay Bob $30
- Bank actions
  1. Does Alice have $30?
  2. Give $30 to Bob
  3. Take $30 from Alice

#### B. Alice to pay Bill $30
- Bank actions
  1. Does Alice have $30?
  2. Give $30 to Bill
  3. Take $30 from Alice

- If Alice starts with $40, can Bob and Bill both get $30?
Race conditions in real life

• E.g., check-then-act at the bank
  ▪ The "debit-credit problem"

_Alice, Bob, Bill, and the Bank_

• A. _Alice_ to pay _Bob_ $30
  ▪ Bank actions
    1. Does Alice have $30?
    2. Give $30 to _Bob_
    3. Take $30 from _Alice_

• B. _Alice_ to pay _Bill_ $30
  ▪ Bank actions
    1. Does Alice have $30?
    2. Give $30 to _Bill_
    3. Take $30 from _Alice_

• If _Alice_ starts with $40, can _Bob_ and _Bill_ both get $30?
Race conditions in *your* real life

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

```java
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:

\texttt{int \ i = 7;}

Thread A:

\texttt{i = 42;}

Thread B:

\texttt{ans = i;}

- What are the possible values for \texttt{ans}?
Some actions are atomic

Precondition:  
\[ \text{int } i = 7; \]

Thread A:  
\[ i = 42; \]

Thread B:  
\[ \text{ans} = i; \]

- What are the possible values for \text{ans}?

\[ i: 00000...00000111 \]

\[ : \]

\[ i: 00000...00101010 \]
Some actions are atomic

Precondition:

\[
\text{int } i = 7;
\]

Thread A:

\[
i = 42;
\]

Thread B:

\[
\text{ans } = i;
\]

- What are the possible values for \text{ans}?

\[
i: \quad 00000\ldots00000111
\]

\[
i: \quad 00000\ldots00101010
\]

- In Java:
  - Reading an int variable is atomic
  - Writing an int variable is atomic

- Thankfully, \text{ans}:

\[
00000\ldots00101111
\]

is not possible
Bad news: some simple actions are not atomic

- Consider a single 64-bit long value

<table>
<thead>
<tr>
<th>high bits</th>
<th>low bits</th>
</tr>
</thead>
</table>

- Concurrently:
  - Thread A writing high bits and low bits
  - Thread B reading high bits and low bits

Precondition: \( \text{long } i = 10000000000; \)

Thread A: \( i = 42; \)

Thread B: \( \text{ans } = i; \)

\[ \text{ans: } 01001\ldots00000000 \]

\[ \text{ans: } 00000\ldots00101010 \]

\[ \text{ans: } 01001\ldots00101010 \]

\( (100000000000) \)

\( (42) \)

\( (100000000042 \text{ or } \ldots) \)
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:
  ```java
  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

```java
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

```
a -> b
b -> c
c -> f
f -> g
h -> i
i -> e
e -> d
d -> a
```
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
Concurrent 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 \textit{happens-before} relation

- Java guarantees a transitive, consistent order for some memory accesses
  - Within a thread, one action \textit{happens-before} another action based on the usual program execution order
  - Release of a lock \textit{happens-before} acquisition of the same lock
  - \texttt{Object.notify} \textit{happens-before} \texttt{Object.wait} returns
  - \texttt{Thread.start} \textit{happens-before} any action of the started thread
  - Write to a \texttt{volatile} field \textit{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 \textit{happens-before} relation
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.atomic package
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...

![Diagram of ConcurrentHashMap](image_url)
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