Principles of Software Construction: Objects, Design and Concurrency

The Perils of Concurrency

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

Jonathan Aldrich

Charlie Garrod
• Homework 4c due tonight

• Homework 5 coming soon
  ▪ Must select partner(s) by tonight (29 October)
  ▪ 5a due next Wednesday (06 November)
  ▪ 5b due the following Tuesday (12 November)
  ▪ 5c due the following Tuesday (19 November)

• Final exam is Monday 09 December, 8:30 – 11:30 a.m.
Key topics from last Thursday
Today: Concurrency, part 1

• The backstory
  ▪ Motivation, goals, problems, ...

• Basic concurrency in Java
  ▪ Synchronization

• Coming soon (but not today):
  ▪ Higher-level abstractions for concurrency
    • Data structures
    • Computational frameworks
Processor speeds over time
Power requirements of a CPU

- Approx.: Capacitance $\times$ Voltage$^2$ $\times$ Frequency

- 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
One option: fix the symptom

- Dissipate the heat
One option: fix the symptom

- **Better:** Dissipate the heat with liquid nitrogen
  - Overclocking by Tom's Hardware's 5 GHz project

http://www.tomshardware.com/reviews/5-ghz-project,731-8.html
Another option: fix the underlying problem

- Reduce heat by limiting power input
  - Adding processors increases power requirements linearly with performance
  - Reduce power requirement by reducing the frequency and voltage
  - Problem: requires concurrent processing
Aside: Three sources of disruptive innovation

- **Growth crosses some threshold**
  - e.g., Concurrency: ability to add transistors exceeded ability to dissipate heat

- **Colliding growth curves**
  - Rapid design change forced by jump from one curve onto another

- **Network effects**
  - Amplification of small triggers leads to rapid change
Aside: The threshold for distributed computing

• Too big for a single computer?
  ▪ Forces use of distributed architecture
  ▪ Shifts responsibility for reliability from hardware to software
  ▪ Allows you to buy cheap flaky machines instead of expensive somewhat-flaky machines
    – Revolutionizes data center design
Aside: Network effects

• Metcalfe's rule: network value grows quadratically in the number of nodes
  ▪ a.k.a. Why my mom has a Facebook account
  ▪ $n(n-1)/2$ potential connections for $n$ nodes

• Creates a strong imperative to merge networks
  • Communication standards, USB, media formats, ...
Concurrency

• Simply: doing more than one thing at a time
  ▪ In software: more than one point of control
    • Threads, processes

• Resources simultaneously accessed by more than one thread or process
Concurrency then and now

- In the past multi-threading was just a convenient abstraction
  - GUI design: event threads
  - Server design: isolate each client's work
  - Workflow design: producers and consumers

- Now: must use concurrency for scalability and performance
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
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?
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
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 \( N \))
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
  void run();

• The java.lang.Thread class
  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++;
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:
  ```java
  classData++;
  ```

  Thread B:
  ```java
  classData++;
  ```

  One possible interleaving of actions:
  1A. Load data(41) from `classData`
  1B. Load data(41) from `classData`
  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:  
\[
\text{int } i = 7;
\]

Thread A:  
\[
i = 42;
\]

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

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

<table>
<thead>
<tr>
<th>Precondition:</th>
<th>Thread A:</th>
<th>Thread B:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int i = 7;</code></td>
<td><code>i = 42;</code></td>
<td><code>ans = i;</code></td>
</tr>
</tbody>
</table>

- **What are the possible values for `ans`?**

  - `i:`
    - `00000...00000111`
    - `...`
    - `00000...00101010`

---

15-214 Garrod
Some actions are atomic

Precondition:

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

Thread A:

\[
i = 42; \\
\]

Thread B:

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

• What are the possible values for \( \text{ans} \)?

\[
i: \quad 00000...00000111 \\
\]

\[
\vdots \\
\]

\[
i: \quad 00000...00101010 \\
\]

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

  • Thankfully, \( \text{ans} : 00000...00101111 \) 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:  
```
long i = 10000000000;
```

Thread A:  
```
i = 42;
```

Thread B:  
```
ans = i;
```

```
ans: 01001...00000000
```

```
ans: 00000...00101010
```

```
ans: 01001...00101010
```

(100000000000)  
(42)  
(100000000042 or ...)
Thursday:

- More concurrency