Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Concurrency

Introduction to concurrency

Josh Bloch  Charlie Garrod
Administrivia

- HW 5a due 9am tomorrow
- Presentations in recitation tomorrow
- Reading due today, Java Concurrency In Practice, Sections 11.3-4
- Midterm 2 has been graded; Grades will be released after class
Key concepts from last Thursday
Challenges of working as a team: Aligning expectations

- How do we make decisions?
Use simple **branch-based development**

Create a new branch for each feature.
- allows parallel development
- no dealing with half-finished code
- no merge conflicts!

Every commit to “master” should pass your CI checks.
Today’s lecture: concurrency motivation and primitives

• Why concurrency?
  – Motivation, goals, problems, ...

• Concurrency primitives in Java

• Coming soon (not today):
  – Higher-level abstractions for concurrency
  – Program structure for concurrency
  – Frameworks for concurrent computation
Moore’s Law (1965) – number of transistors on a chip doubles every two years
CPU Performance and Power Consumption

• **Dennard Scaling** (1974) – each time you double transistor density:
  – Speed (frequency) goes up by about 40% (Why?)
  – While power consumption of the chip stays constant (proportional to area)
• Combined w/ Moore’s law, every 4 years the number of transistors quadruples, speed doubles, and power consumption stays constant
• It was great while it lasted
  – Came to a grinding halt around 2004 due to *leakage currents* 😞
  – More power required at higher frequency, generating more heat
  – There’s a limit to how much heat a chip can tolerate
One option: fix the symptom

• Dissipate the heat
One option: fix the symptom

• Better(?): Dissipate the heat with liquid nitrogen
42 Years of Microprocessor Trend Data

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp
Concurrency then and now

• In the past, multi-threading just a convenient abstraction
  – GUI design: event dispatch thread
  – Server design: isolate each client's work
  – Workflow design: isolate producers and consumers

• Now: required for scalability and performance
We are all concurrent programmers

- Java is inherently multithreaded
- To utilize modern processors, we must write multithreaded code
- Good news: a lot of it is written for you
  - Excellent libraries exist (e.g., java.util.concurrent)
- Bad news: you still must understand fundamentals
  - ...to use libraries effectively
  - ...to debug programs that make use of them
Aside: Concurrency vs. parallelism, visualized

- Concurrency without parallelism:

- Concurrency with parallelism:
Basic concurrency in Java

• An interface representing a task
  public interface Runnable {
    void run();
  }

• A class to execute a task in a CPU thread
  public class Thread {
    public Thread(Runnable task);
    public void start();
    public void join();
    ...
  }
Example: Money-grab (1)

```java
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance += amount;
    }

    public long balance() {
        return balance;
    }
}
```
Example: Money-grab (2)

What would you expect this program to print?

```java
public static void main(String[] args) throws InterruptedException {
    BankAccount bugs = new BankAccount(100);
    BankAccount daffy = new BankAccount(100);

    Thread bugsThread = new Thread(() -> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(daffy, bugs, 100);
    });

    Thread daffyThread = new Thread(() -> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(bugs, daffy, 100);
    });

    bugsThread.start(); daffyThread.start();
    bugsThread.join(); daffyThread.join();
    System.out.println(bugs.balance() + daffy.balance());
}
```
What went wrong?

• Daffy & Bugs threads had a race condition for shared data
  – Transfers did not happen in sequence
• Reads and writes interleaved randomly
  – Random results ensued
The challenge of concurrency control

• Not enough concurrency control: *safety failure*
  – Incorrect computation

• Too much concurrency control: *liveness failure*
  – Possibly no computation at all (*deadlock* or *livelock*)
Shared mutable state requires concurrency control

• Three basic choices:
  1. Don't mutate: share only immutable state
  2. Don't share: isolate mutable state in individual threads
  3. If you must share mutable state: synchronize to achieve safety
An easy fix:

```java
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static synchronized void transferFrom(BankAccount source, BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance += amount;
    }

    public long balance() {
        return balance;
    }
}
```
Concurrency control with Java’s *intrinsic* locks

• `synchronized (lock) { ... }`
  – Synchronizes entire block on object `lock`; cannot forget to unlock
  – Intrinsic locks are *exclusive*: One thread at a time holds the lock
  – Intrinsic locks are *reentrant*: A thread can repeatedly get same lock
Concurrency control with Java’s *intrinsic* locks

- **synchronized** (lock) { ... }
  - Synchronizes entire block on object lock; cannot forget to unlock
  - Intrinsic locks are *exclusive*: One thread at a time holds the lock
  - Intrinsic locks are *reentrant*: A thread can repeatedly get same lock

- **synchronized** on an instance method
  - Equivalent to synchronized (this) { ... } for entire method

- **synchronized** on a static method in class Foo
  - Equivalent to synchronized (Foo.class) { ... } for entire method
Another example: serial number generation

*What would you expect this program to print?*

```java
public class SerialNumber {
    private static long nextSerialNumber = 0;

    public static long generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads)
            thread.join();
        System.out.println(generateSerialNumber());
    }
}
```
What went wrong?

• 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

• Java’s `++` (increment) operator is not atomic!
  – It reads a field, increments value, and writes it back

• If multiple calls to `generateSerialNumber` see the same value, they generate duplicates
Again, the fix is easy

```java
public class SerialNumber {
    private static long nextSerialNumber = 0;

    public static synchronized long generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads)
            thread.join();
        System.out.println(generateSerialNumber());
    }
}
```
But you can do better!

`java.util.concurrent is your friend`

```java
public class SerialNumber {
    private static AtomicLong nextSerialNumber = new AtomicLong();
    public static long generateSerialNumber() {
        return nextSerialNumber.getAndIncrement();
    }

    public static void main(String[] args) throws InterruptedException{
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
```
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:
\[
\text{int } i = 7;
\]

Thread A:
\[
i = 42;
\]

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

• What are the possible values for ans?

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

\[
\vdots
\]

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

\[
\text{ans: } \quad 00000...00101111
\]
Some actions are atomic

Precondition: 
\[
\text{int } i = 7;
\]
Thread A: 
\[
i = 42;
\]
Thread B: 
\[
\text{ans } = i;
\]

- What are the possible values for ans?

- 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

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

Precondition:

- \( \text{long } i = 10\_000\_000\_000; \)
- \( i = 42; \)
- \( \text{ans } = i; \)

\[
\begin{align*}
\text{Thread A:} & \quad \text{ans: } 01001\ldots00000000 \\
\text{Thread B:} & \quad (10,000,000,000) \\
\text{Thread A:} & \quad \text{ans: } 00000\ldots00101010 \\
\text{Thread B:} & \quad (42) \\
\text{Thread A:} & \quad \text{ans: } 01001\ldots00101010 \\
\text{Thread B:} & \quad (10,000,000,042)
\end{align*}
\]

All are possible!
Yet another example: cooperative thread termination

*How long would you expect this program to run?*

```java
public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```
What went wrong?

• In the absence of synchronization, there is no guarantee as to when, **if ever**, one thread will see changes made by another.

• **JVMs can and do perform this optimization (“hoisting”):**
  
  ```java
  while (!done)
      /* do something */ ;
  ```

  becomes:

  ```java
  if (!done)
      while (true)
          /* do something */ ;
  ```
Why is synchronization required for communication among threads?

- **Naively:**
  - Thread state shared in memory

- **A (slightly) more accurate view:**
  - Separate state stored in registers and caches, even if shared
How do you fix it?

public class StopThread {
    private static boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }
    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested()) {
                /* Do something */
            }
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(10);
        requestStop();
    }
}
A better(?) solution
volatile is synchronization without mutual exclusion

```java
class StopThread {  
private static volatile boolean stopRequested;

public static void main(String[] args) throws Exception {
    Thread backgroundThread = new Thread(() -> {
        while (!stopRequested)
            /* Do something */;
    });
    backgroundThread.start();

    TimeUnit.SECONDS.sleep(10);
    stopRequested = true;
}
```

Summary

• Like it or not, you’re a concurrent programmer
• Ideally, avoid shared mutable state
  – If you can’t avoid it, synchronize properly
• Even atomic operations require synchronization
  – e.g., stopRequested = true
• Some things that look atomic aren’t (e.g., val++)