Zigurd **MEDNIEKS**
Series Editor

ANDROID™
DEEP
DIVE

G. Blake **MEIKE**

ANDROID™ CONCURRENCY

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Android™ Concurrency

About the Android Deep Dive Series

Zigurd Mednieks, Series Editor

The Android Deep Dive Series is for intermediate and expert developers who use Android Studio and Java, but do not have comprehensive knowledge of Android system-level programming or deep knowledge of Android APIs. Readers of this series want to bolster their knowledge of fundamentally important topics.

Each book in the series stands alone and provides expertise, idioms, frameworks, and engineering approaches. They provide in-depth information, correct patterns and idioms, and ways of avoiding bugs and other problems. The books also take advantage of new Android releases, and avoid deprecated parts of the APIs.

About the Series Editor

Zigurd Mednieks is a consultant to leading OEMs, enterprises, and entrepreneurial ventures creating Android-based systems and software. Previously he was chief architect at D2 Technologies, a voice-over-IP (VoIP) technology provider, and a founder of OpenMSobile, an Android-compatibility technology company. At D2 he led engineering and product definition work for products that blended communication and social media in purpose-built embedded systems and on the Android platform. He is lead author of *Programming Android* and *Enterprise Android.*

Android™ Concurrency

G. Blake Meike

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi
Mexico City • São Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016937763

ISBN-13: 978-0-13-417743-4
ISBN-10: 0-13-417743-6

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana
First printing: June 2016

Editor-in-Chief
Greg Wiegand

Executive Editor
Laura Lewin

Development Editor
Sheri Replin

Managing Editor
Sandra Schroeder

Project Editor
Lori Lyons

Project Manager
Ellora Sengupta

Copy Editor
Abigail Manheim Bass

Indexer
Cheryl Lenser

Proofreader
Natarajan

Technical Reviewers
Joe Bowbeer
Thomas Kountis

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
codeMantra

Graphics Conversion
Vived Graphics

❖

This book is for my mother, Sally Ann Obermeyer Meike, who encouraged me to be a snoot, long before David Foster Wallace re-purposed the term

❖

This page intentionally left blank

Contents at a Glance

This page intentionally left blank

Table of Contents

Preface

I have had the opportunity during my years in working in this industry, to see concurrency in many contexts. When I was in school, it was a topic for dissertations. As a journeyman developer, I saw a lot of concurrent code, nearly all of it Java, in distributed back-end systems. Recently, I've had the opportunity to experience first-hand the turn to languages like Erlang and Scala, in the hope of making concurrent code easier to design and write.

I recall early in my career, being coached by a very supportive interviewer into reinventing double-checked locking. I certainly recall the furor, about a year later, when it was discovered that the double-checked locking idiom was not safe, and soon after, the first sighting of incorrect byte-code in the wild. Perhaps most surprising of all, though, I recently removed an implementation of the double-checked locking idiom from a piece of code written in 2015.

The constant, over this time, is the mystery and debate surrounding the topic. Perfectly competent novices suddenly balk or make naïve mistakes when concurrent code is necessary. Developers who are otherwise entirely reasonable sometimes disagree—occasionally quite vehemently—over the correctness of a particular piece of concurrent code. Their arguments, which may go on for hours, inevitably end up hinging on minutiae so fragile that the argument's actual winner is irrelevant.

I readily admit to feeling some kind of glee when I walked out of that early interview after having been led to double-checked locking. It was a shibboleth. I'd been initiated! Recently, I think I've seen the same kind of excitement in the faces of lecturers and their audiences as the lecturer passes on a secret: some fast and loose—and frequently downright incorrect—concurrency trick.

A really clever algorithm and a good shot of glee are wonderful things. We would all write better code, though, if we could strip some of the mystery and magic from concurrent programming. It would be great if instead of being the realm of the wizard, the correctness of a piece of concurrent code were something on which the opinions of two developers—even two developers with wildly different interests—rapidly converged.

Who Should Read This Book

This book is intended for developers with some experience with Android development.

If you are a novice developer, you will probably find some of the terms and concepts here unfamiliar. If you are a developer working on his or her very first Android app, you will probably be more concerned with simply getting familiar with the Android framework.

There are some really good books for you, already in existence, if you fit into either of these two categories. If that is the case, I encourage you to set this book aside for a while, enjoy the thrill of the steep part of a learning curve, and to come back when you have one complete Android app under your belt.

While it may sound obvious, I intend this book for reading. It is neither a cookbook nor a reference manual. I absolutely encourage you to try out the sample code. The examples are just sandboxes for experiments. Extend them. Try new experiments with them. Build your own personal understanding of the details of the Android OS. I hope, though, that you don't have to prop the book open next to your laptop to get value from it.

I was at one time bewitched by Perl. Looking back, I think that a major reason for that is that it was so much fun to read the Camel book (not the pink one, the blue one: the 2nd edition). I recall many enjoyable hours far from a keyboard and with no specific application in mind, simply reading that book.

I do not compare myself to Larry Wall. I do not by any means compare Android to Perl. I do hope, though, that you enjoy just reading this book. I hope that it is something that will make good company, perhaps on a plane flight or a long commute.

How This Book Is Organized

The first three chapters of this book will be a review for the audience for whom the book is intended.

I urge you not to skip Chapter 1, at least. It presents a model of concurrency that is somewhat atypical and that is the basis for the discussion in the following chapters.

Chapters 2 and 3 are intentionally short. They are a refresher for some basic ideas and provide an opportunity to reintroduce some common idioms. Experienced developers may choose to skim them.

Chapter 4 is a cautionary tale.

The heart of the book is Chapters 5 through 7. These chapters are a deep dive into some of the details of the Android operating system.

Chapter 8 is dessert: a bit of a how-to for some concurrency tools.

Example Code

Most of the code shown in examples in this book can be found on GitHub at https://github.com/AndroidConcurrencyDeepDive. If you experiment with them and discover something interesting or amusing, by all means submit a pull-request to share it with others.

Register your copy of *Android Concurrency* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN 9780134177434 and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products."

Conventions Used in This Book

The following typographical conventions are used in this book:

- **Bold** indicates new terms, URLs, email addresses, filenames, and file extensions.
- `Constant width` is used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.
- **`Constant width bold`** shows commands or other texts that should be typed by the user.
- *`Constant width italic`* shows texts that should be replaced with the user-supplied values or with the values determined by the context.

> **Note**
>
> A Note signifies a tip, suggestion, or general note.

About the Author

Blake Meike is a passionate engineer, architect, and code poet. As an author, speaker, and instructor, he has taught thousands of people how to write Android apps that aren't toys. He has more than 20 years of coding experience, most of it with Java, building systems as large as Amazon's massively scalable AutoScaling service and as small as a pre-Android OSS Linux/Java-based platform for cell phones. He is co-author of several other books, including O'Reilly's bestselling "Programming Android" and Wiley's "Enterprise Android." Blake holds a degree in Mathematics and Computer Science from Dartmouth College and was a founding member of Twitter University. He lives in Oakland, CA, and works for Cyanogen Inc.

Acknowledgements

This book owes its existence to a long list of contributors. My longtime colleague, Zigurd Mednieks, proposed it. Laura Lewin, Executive Editor at Pearson Technology Group, gave it a green light, and Carol Jelen, Literary Agent at Jelen Publishing, put a deal together.

The production staff at Pearson deftly turned the manuscript into a book. Illustrator Jenny Huckleberry cleaned up the figures and Development Editor Sheri Replin, Copy Editor Abigail Manheim Bass, and Project Editor Lori Lyons cleaned up the drafts. Editorial Assistant Olivia Basegio and Project Manager Ellora Sengupta kept us all together through the process. My thanks to all of you.

A special "thank you" and my eternal gratitude go to Laura Lewin, Executive Editor. Her patience with my complete inability to stick to any kind of schedule surely qualifies her for immediate sainthood.

On this book, my first as a solo author, my technical editors Joe Bowbeer, Thomas Kountis, and Zigurd Mednieks contributed so much that I am tempted to list them as co-authors. They were everything that I could have hoped for in a technical review team. Each, a recognized expert in the field, took the time to read carefully, understand, and then suggest changes as small as rewording and as big as re-thinking. While any remaining errors are strictly my own, any clarity and accuracy are due to them. I am incredibly lucky to have had their help. Thanks, guys!

Finally and always, thanks to my wonderful wife, Catherine, who endured yet another year of weekends with her husband on the couch, headphones on, incommunicado. That glitter heart is still for you, babe!

This page intentionally left blank

1

Understanding Concurrency

We propose to use the delays τ as absolute units of time which can be relied upon to synchronize the functions of various parts of the device.

John von Neumann

In order to build correct, concurrent Android programs, a developer needs a good model of concurrent processes, how they work, and what they are for. Concurrency actually isn't a big deal for most normal humans. For any multi-celled animal—arguably even for viruses—it is just normal existence. It is only those of us obsessed with computers that give a second thought to the idea of walking and chewing gum at the same time.

Concurrency Made Hard

Walking and chewing gum isn't easy in the strange world of Dr. John von Neumann. In his 1945 paper, "The First Draft Report on the EDVAC" (von Neumann 1954), he describes the architecture of one of the very first electronic digital computers. In most ways, that architecture has changed very little in the seventy years since. Throughout their history, digital computers have been, roughly speaking, gigantic balls of state that are transformed, over time, by a sequence of well-defined operations. Time and order are intrinsic parts of the definition of the machine.

Most of computer science has been the discussion of clever sequences of operations that will transform one machine state into another, more desirable, state. As modern machines commonly have more than 10^{14} possible states, those discussions are already barely manageable. If the order in which transformations take place can vary, the discussion necessarily broadens to include all possible combinations of all possible states, and becomes utterly impossible. Sequential execution is the law of the land.

Concurrency in Software

Of course, computer languages are written for humans. They are intended to help us express an algorithm (the sequence of instructions that transforms the machine state) efficiently, correctly, and, perhaps, even in a way that future human readers can understand.

Early programming languages were, essentially, an extension of the hardware. Even today many are reflections of the machine architecture they were originally designed to control. Nearly all of them are **procedural** and consist of lists of instructions for changing (**mutating**) the state of memory. Because it is simply too difficult to reason about all of the possible states of that memory, languages have, over time, become more and more restrictive about the state changes they allow a developer to express. One way to look at the history of programming language design is as a quest for a system that allows developers to express correct algorithms easily, and not express incorrect ones at all.

The very first languages were machine languages—code that translated, one-for-one, into instructions for the computer. These languages were undesirable for two reasons. First, expressing even a very simple idea might take tens of lines of code. Second, it was much too easy to express errors.

Over time, in order to restrict and manage the ways in which a program could change state, languages have narrowed the choices. Most, for instance, restrict program execution from arbitrarily skipping around between instructions to using now-familiar conditionals, loops, and procedure calls. Modules and eventually OOP (Object-Oriented Programming) followed, as ways of separating a program into small, understandable pieces and then limiting the way those pieces can interact. This modularized, building-block approach makes modern languages more abstract and expressive. Some even have well-developed type systems that help prevent errors. Almost all of them, though, are still imperative: lists of instructions for changing machine state.

Functional Programming

While most computer research and development focused on doing more and more complicated things, on bigger and faster hardware based on von Neumann architecture, a small but persistent contingent has pursued a completely different idea: *functional programming*.

A purely functional program differs from a procedural program in that it does not have mutable state. Instead of reasoning about successive changes to machine state, functional languages reason about evaluating functions at given parameters. This is a fairly radical idea and it takes some thinking to understand how it could work. If it were possible, though, it would have some very appealing aspects from a concurrency point of view. In particular, if there is no mutable state, there is no implicit time or order. If there is no implicit order, then concurrency is just an uninteresting tautology.

John McCarthy introduced Lisp, the first functional language, in 1958, only a year or two after the creation of the first commonly accepted procedural language, Fortran. Since then, Lisp and its functional relatives (Scheme, ML, Haskel, Erlang, and so on) have been variously dismissed as brilliant but impractical, as educational tools, or as shibboleths for hipster developers. Now that Moore's law (Moore, 1965) is more likely to predict the number of processors on a chip

than the speed of a single processor, people are not dismissing functional languages anymore. (By 1975, Moore formalized this concept when he revised his original thoughts, and said the number of integrated circuit (IC) components would double every two years.)

Programming in a functional style is an important strategy for concurrent programming and may become more important in the future. Java, the language of Android, does not qualify as a functional language and certainly does not support the complex type system associated with most functional languages.

Language as Contract

Functional or procedural, a programming language is an abstraction. Only a tiny fraction of developers need to get anywhere near machine language these days. Even that tiny fraction is probably writing code in a virtual instruction set, implemented by a software virtual machine, or by chip firmware. The only developers likely to understand the precise behavior of the instruction set for a particular piece of hardware, in detail, are the ones writing compilers for it.

It follows that a developer, writing a program in some particular language, expects to understand the behavior of that program by making assertions in that language. A developer reasons in the language in which the program is written—the abstraction—and almost never needs to demonstrate that a program is correct (or incorrect) by examining the actual machine code. She might reason, for instance, that something happens 14 times because the loop counter is initialized to 13, decremented each time through the loop, and the loop is terminated when the counter reaches 0.

This is important because most of our languages are imperative (not functional) abstractions. Even though hardware, registers, caches, instructions pipelines, and clock cycles typically don't come up during program design, when we reason about our programs we are, nonetheless, reasoning about sequence.

Concurrency in Hardware

It is supremely ironic that procedural languages, originally reflections of the architecture they were designed to control, no longer represent the behavior of computer hardware. Although the CPU of an early computer might have been capable of only a single operation per tick of its internal clock, all modern processors are performing multiple tasks simultaneously. It would make no sense at all to idle even a quarter of the transistors on a 4-billion gate IC while waiting for some single operation to complete.

> **Why Is Everything Sequential?**
>
> It is possible that sequential execution is just an inherent part of how we humans understand things. Perhaps we first imposed it on our hardware design and then perpetuated it in our language design because it is a reflection of the way our minds work.

Hardware is physical stuff. It is part of the real world, and the real world is most definitely not sequential. Modern hardware is very parallel.

In addition to running on parallel processors, modern programs are more and more frequently interacting with a wildly parallel world. The owners of even fairly ordinary feature-phones are constantly multitasking: listening to music while browsing the web, or answering the phone call that suddenly arrives. They expect the phone to keep up. At the same time, sensors, hardware buttons, touch screens, and microphones are all simultaneously sending data to programs. Maintaining the illusion of "sequentiality" is quite a feat.

A developer is in an odd position. As shown in Figure 1.1, she is building a set of instructions for a sequential abstraction that will run on a highly parallel processor for a program that will interact with a parallel world.

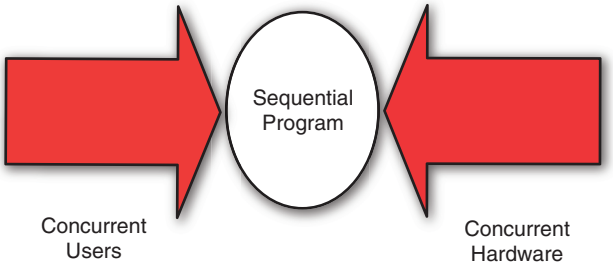

Figure 1.1 A sequential program in a concurrent world

Concurrency Made Simple

The purpose of the discussion, up to this point, has been to reframe the idea of concurrency. Concurrency is not a way to make a program run faster. It is not a complex juggling trick that ninja coders use to keep multiple balls in the air at one time. On the contrary, it is the apparent sequential execution of a program that is the complex trick. Sequential execution is an illusion maintained by a cabal of compiler writers and hardware architects. Concurrency is simply the relaxation of a fabricated constraint.

Threads

In the developer's environment, where time and order are rigid and implicit constraints, "concurrency" is just another word for "order unspecified"—the way things are everywhere else. A concurrent program is just one that announces to an already-concurrent world that its correctness does not depend on the ordering of events that occur in separate components. In a concurrent program, those separate, partially ordered components are called **threads of execution**, or just **threads**. Within a thread, instructions are still executed in rigidly sequential order. The order of execution of instructions in two separate threads, however, is completely unspecified.

In the early days of computing, the choice of threads as a model for concurrency was not obvious. Developers that needed out-of-order processing were forced to brew their own

concurrency constructs. Both literature and code from the 1960s contain a wide variety of models for asynchronous execution.

Threads probably originated in the late 1960s, with IBM's OS/360. They were called "tasks," and were an OS-level service that saved developers the trouble of building their own concurrency abstraction. In 1991 Java, called Oak at the time, adopted the thread model and supported it in the language, even on operating systems that did not.

Even today, threads are not the only model for concurrency. Languages such as Erlang, Go, and Clojure, for instance, each use an entirely different model.

Introducing threads into the programming model does not present an intrinsic problem. Operating two cars in parallel causes no problems unless they both try to occupy the same space at the same time. Similarly, operating two threads that are completely independent is also perfectly safe. There are millions of programs, each concurrently running in its own thread of execution on millions of separate computers, at this very moment. Most of these programs don't interact with each other in any way and their behavior is perfectly well defined. The problems only arise when threads need to share state and resources.

Atomic Execution

When multiple threads change state that is accessible to both, the results can easily be nondeterministic. Because there is no relationship between the order in which statements are executed, subtle changes in timing can change the result of running the program.

Consider the following code:

```
executionCount++;
someTask();
```

Just by inspection, it seems likely that the variable `executionCount` is meant to count the number of times that the function someTask is called. In a concurrent environment, however, this code, as it stands, does not have deterministic behavior because the ++ operation is not atomic—it is not a single, indivisible action. Table 1.1 demonstrates an execution sequence that fails to record one execution.

Table 1.1 **Non-Atomic Execution**

executionCount = 4	
Thread 1	**Thread 2**
read execution count (4)	read execution count (4)
increment (5)	increment (5)
store execution count (5)	store execution count (5)
call someTask	call someTask

Synchronization is the basic mechanism through which multiple Java threads share state in such a way that the result of the interaction is deterministic. In itself, synchronization is a simple idea: a **critical section** of code is protected by a mutual-exclusion lock or **mutex**. When a thread enters the critical section—that is, it begins executing instructions from it—it is said to seize the mutex. No other thread can enter the section until the first thread leaves.

The previous code becomes deterministic if only a single thread is allowed to execute the critical section at any given time:

```
synchronized(this) {
    executionCount++;
    someTask();
}
```

Synchronization is the crux of creating correct concurrent Java programs, and is the basis for a lot of things that are definitely not simple. Those things make up the content of the rest of this book.

Visibility

There is one more thing, however, that is simple. Remember that the previous example is an abstraction! It is written in a computer language—Java, in this case—and is, therefore, related to the actual behavior of hardware only by the grace of compiler writers, JVM developers, and hardware architects. Those two Java statements translate into hundreds of microinstructions, many of them executed in parallel, over tens of hardware clock cycles. The illusion that there are two statements, happening in order, is no more than an illusion.

Maintaining the illusion is not something that near-the-metal developers do naturally. On the contrary, they find sequential programs naive, clumsy, and wasteful. They are only too happy to fix them by re-ordering instructions, executing multiple instructions in parallel, representing a single piece of program state as multiple copies, and so on. By doing so, they do their very best to make optimal use of the immense power of the multiple processors that comprise even the tiny devices we carry in our pockets.

In general, we're glad to have them perform these optimizations. They make our programs run faster, on multiple hardware platforms, using tricks in which application developers are just not that interested. There is one important condition on this optimization, however: *They must not break the illusion of sequentiality!* In other words, compilers and hardware pipelines can reorder and parallelize all they want to optimize the code, as long as developers can't tell that they did it.

In making a program concurrent, a developer clearly states that there is no sequential dependency between the states controlled by different threads. If there is no sequential dependency, a compiler should be free to perform all sorts of optimizations that would otherwise have been unsafe. Without an explicit ordering between events in different threads, the compiler is free to make changes in the execution sequence of one thread without any consideration of the statements in any other.

A correct concurrent program is one that abides by the contract for maintaining an illusion. Negotiation between application programmers and hardware developers produce a language, and that language is a contract. The application developers get their illusion of sequential execution, something about which they can reason. The hardware developers get a toolbox of clever tricks they can use to make programs run fast. In the middle is the contract.

In Java, that contract is called the **memory model**. On one side of the contract is the application programmer, reasoning about her program in a high-level language. On the other side of the contract are the compiler writers, virtual machine developers, and hardware architects, moving everything that isn't explicitly forbidden. Developers who talk about hardware when discussing concurrency are missing the point. A correct concurrent program is not about hardware; it is about developers keeping their end of the contract.

Fortunately, in Java, the contract is easily stated. The following single sentence states it almost completely:

> Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronization.
>
> (Göetz, et al. 2006)

A correct concurrent Java program is one that abides by this contract—no more, no less. Note in particular that whether a thread reads or writes mutable state does not affect its need for synchronization in any way.

Summary

Concurrency itself is nothing to be scared of. We all deal with it in the real world, all day, every day. What is difficult is writing concurrent programs for computers based on mutable state. It is difficult because the concept of order is such an important implicit basis for the way we reason about our programs.

Concurrency is the relaxation of the rigid order inherent in imperative computer languages. Java's mechanism for doing this is the thread. A thread executes instructions in an order that is not related to the order of execution of instructions in other threads. Developers use mutual exclusion locks (mutexes) to control thread access to critical sections of code, thereby limiting the number of ways that two different threads can execute instructions in the section.

Most of today's computer languages manufacture an illusion of sequential execution. Behind the scenes, however, they fiercely reorder, parallelize, and cache to make the best use of hardware. The only thing that prevents those optimizations from making a program behave non-deterministically, is a contract. A correct program is one that abides by that contract.

No one ever said that concurrency was easy. It is, however, fairly simple. Just follow the contract.

This page intentionally left blank

Index

A

B

C

D

E

F

G

H

I

M

N

O

P

R

S

T

U