The Little Book of Semaphores

Allen B. Downey

Version 2.1.5

The Little Book of Semaphores

Second Edition

Version 2.1.5

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a book, which can be converted to other formats and printed.

This book was typeset by the author using latex, dvips and ps2pdf, among other free, open-source programs. The LaTeX source for this book is available from http://greenteapress.com/semaphores.

Preface

Most undergraduate Operating Systems textbooks have a module on Synchronization, which usually presents a set of primitives (mutexes, semaphores, monitors, and sometimes condition variables), and classical problems like readers-writers and producers-consumers.

When I took the Operating Systems class at Berkeley, and taught it at Colby College, I got the impression that most students were able to understand the solutions to these problems, but few would have been able to produce them, or solve similar problems.

One reason students don't understand this material deeply is that it takes more time, and more practice, than most classes can spare. Synchronization is just one of the modules competing for space in an Operating Systems class, and I'm not sure I can argue that it is the most important. But I do think it is one of the most challenging, interesting, and (done right) fun.

I wrote the first edition this book with the goal of identifying synchronization idioms and patterns that could be understood in isolation and then assembled to solve complex problems. This was a challenge, because synchronization code doesn't compose well; as the number of components increases, the number of interactions grows unmanageably.

Nevertheless, I found patterns in the solutions I saw, and discovered at least some systematic approaches to assembling solutions that are demonstrably correct.

I had a chance to test this approach when I taught Operating Systems at Wellesley College. I used the first edition of *The Little Book of Semaphores* along with one of the standard textbooks, and I taught Synchronization as a concurrent thread for the duration of the course. Each week I gave the students a few pages from the book, ending with a puzzle, and sometimes a hint. I told them not to look at the hint unless they were stumped.

I also gave them some tools for testing their solutions: a small magnetic whiteboard where they could write code, and a stack of magnets to represent the threads executing the code.

The results were dramatic. Given more time to absorb the material, students demonstrated a depth of understanding I had not seen before. More importantly, most of them were able to solve most of the puzzles. In some cases they reinvented classical solutions; in other cases they found creative new approaches.

When I moved to Olin College, I took the next step and created a half-class, called Synchronization, which covered *The Little Book of Semaphores* and also the implementation of synchronization primitives in x86 Assembly Language, POSIX, and Python.

The students who took the class helped me find errors in the first edition and several of them contributed solutions that were better than mine. At the end of the semester, I asked each of them to write a new, original problem (preferably with a solution). I have added their contributions to the second edition.

Also since the first edition appeared, Kenneth Reek presented the article "Design Patterns for Semaphores" at the ACM Special Interest Group for Computer Science Education. He presents a problem, which I have cast as the Sushi Bar Problem, and two solutions that demonstrate patterns he calls "Pass the baton" and "I'll do it for you." Once I came to appreciate these patterns, I was able to apply them to some of the problems from the first edition and produce solutions that I think are better.

One other change in the second edition is the syntax. After I wrote the first edition, I learned Python, which is not only a great programming language; it also makes a great pseudocode language. So I switched from the C-like syntax in the first edition to syntax that is pretty close to executable Python[1]. In fact, I have written a simulator that can execute many of the solutions in this book.

Readers who are not familiar with Python will (I hope) find it mostly obvious. In cases where I use a Python-specific feature, I explain the syntax and what it means. I hope that these changes make the book more readable.

The pagination of this book might seem peculiar, but there is a method to my whitespace. After each puzzle, I leave enough space that the hint appears on the next sheet of paper and the solution on the next sheet after that. When I use this book in my class, I hand it out a few pages at a time, and students collect them in a binder. My pagination system makes it possible to hand out a problem without giving away the hint or the solution. Sometimes I fold and staple the hint and hand it out along with the problem so that students can decide whether and when to look at the hint. If you print the book single-sided, you can discard the blank pages and the system still works.

 I hope that people will find this book useful, but I also hope they will help continue to develop it by sending in corrections, suggestions, and additional material. Thanks!

Allen B. Downey
Needham, MA
June 1, 2005

[1]The primary difference is that I sometimes use indentation to indicate code that is protected by a mutex, which would cause syntax errors in Python.

Contributor's list

The following are some of the people who have contributed to this book:

- Many of the problems in this book are variations of classical problems that appeared first in technical articles and then in textbooks. Whenever I know the origin of a problem or solution, I acknowledge it in the text.
- I also thank the students at Wellesley College who worked with the first edition of the book, and the students at Olin College who worked with the second edition.
- Se Won sent in a small but important correction in my presentation of Tanenbaum's solution to the Dining Philosophers Problem.
- Daniel Zingaro punched a hole in the Dancer's problem, which provoked me to rewrite that section. I can only hope that it makes more sense now. Daniel also pointed out an error in a previous version of my solution to the H_2O problem, and then wrote back a year later with some typos.
- Thomas Hansen found a typo in the Cigarette smokers problem.
- Pascal Rütten pointed out several typos, including my embarrassing misspelling of Edsger Dijkstra.
- Marcelo Johann pointed out an error in my solution to the Dining Savages problem, and fixed it!
- Roger Shipman sent a whole passel of corrections as well as an interesting variation on the Barrier problem.
- Jon Cass pointed out an omission in the discussion of dining philosophers.
- Krzysztof Kościuszkiewicz sent in several corrections, including a missing line in the Fifo class definition.
- Fritz Vaandrager at the Radboud University Nijmegen in the Netherlands and his students Marc Schoolderman, Manuel Stampe and Lars Lockefeer used a tool called UPPAAL to check several of the solutions in this book and found errors in my solutions to the Room Party problem and the Modus Hall problem.
- Eric Gorr pointed out an explanation in Chapter 3 that was not exactly right.
- Jouni Leppäjärvi helped clarify the origins of semaphores.
- Christoph Bartoschek found an error in a solution to the exclusive dance problem.
- Eus found a typo in Chapter 3.

- Tak-Shing Chan found an out-of-bounds error in `counter_mutex.c`.
- Roman V. Kiseliov made several suggestions for improving the appearance of the book, and helped me with some LaTeX issues.
- Alejandro Céspedes is working on the Spanish translation of this book and found some typos.
- Erich Nahum found a problem in my adaptation of Kenneth Reek's solution to the Sushi Bar Problem.
- Martin Storsjö sent a correction to the generalized smokers problem.
- Cris Hawkins pointed out an unused variable.

Contents

Chapter 1

Introduction

1.1 Synchronization

In common use, "synchronization" means making two things happen at the same time. In computer systems, synchronization is a little more general; it refers to relationships among events—any number of events, and any kind of relationship (before, during, after).

Computer programmers are often concerned with **synchronization constraints**, which are requirements pertaining to the order of events. Examples include:

Serialization: Event A must happen before Event B.

Mutual exclusion: Events A and B must not happen at the same time.

In real life we often check and enforce synchronization constraints using a clock. How do we know if A happened before B? If we know what time both events occurred, we can just compare the times.

In computer systems, we often need to satisfy synchronization constraints without the benefit of a clock, either because there is no universal clock, or because we don't know with fine enough resolution when events occur.

That's what this book is about: software techniques for enforcing synchronization constraints.

1.2 Execution model

In order to understand software synchronization, you have to have a model of how computer programs run. In the simplest model, computers execute one instruction after another in sequence. In this model, synchronization is trivial; we can tell the order of events by looking at the program. If Statement A comes before Statement B, it will be executed first.

There are two ways things get more complicated. One possibility is that the computer is parallel, meaning that it has multiple processors running at the same time. In that case it is not easy to know if a statement on one processor is executed before a statement on another.

Another possibility is that a single processor is running multiple threads of execution. A thread is a sequence of instructions that execute sequentially. If there are multiple threads, then the processor can work on one for a while, then switch to another, and so on.

In general the programmer has no control over when each thread runs; the operating system (specifically, the scheduler) makes those decisions. As a result, again, the programmer can't tell when statements in different threads will be executed.

For purposes of synchronization, there is no difference between the parallel model and the multithread model. The issue is the same—within one processor (or one thread) we know the order of execution, but between processors (or threads) it is impossible to tell.

A real world example might make this clearer. Imagine that you and your friend Bob live in different cities, and one day, around dinner time, you start to wonder who ate lunch first that day, you or Bob. How would you find out?

Obviously you could call him and ask what time he ate lunch. But what if you started lunch at 11:59 by your clock and Bob started lunch at 12:01 by his clock? Can you be sure who started first? Unless you are both very careful to keep accurate clocks, you can't.

Computer systems face the same problem because, even though their clocks are usually accurate, there is always a limit to their precision. In addition, most of the time the computer does not keep track of what time things happen. There are just too many things happening, too fast, to record the exact time of everything.

Puzzle: Assuming that Bob is willing to follow simple instructions, is there any way you can *guarantee* that tomorrow you will eat lunch before Bob?