The Little Book of Semaphores

Allen B. Downey

Version 2.2.1

2

The Little Book of Semaphores

Second Edition

Version 2.2.1

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 Synchro nization, which usually presents a set of primitives (mutexes, semaphores, mon itors, 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, stu dents 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 Com puter 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 ob vious. 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.

This is a Free Book, which means that anyone is welcome to read, copy, modify and redistribute it, subject to the restrictions of the license. 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 pro tected 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 R¨utten pointed out several typos, including my embarrassing mis spelling 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 Ko´sciuszkiewicz 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 Lepp¨aj¨arvi 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 C´espedes 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 solu tion to the Sushi Bar Problem.

• Martin Storsj¨o sent a correction to the generalized smokers

problem. • Cris Hawkins pointed out an unused variable.

• Adolfo Di Mare found the missing "and".

• Simon Ellis found a typo.

• Benjamin Nash found a typo, an error in one solution, and a malfeature in another.

• Alejandro Pulver found a problem with the Barbershop solution.

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 con

straints, 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 synchro nization 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

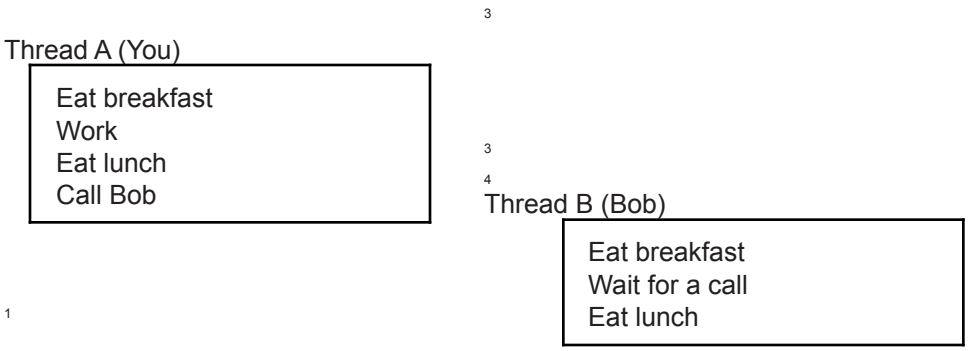
clock? Can you be sure who started first? Unless you are both very careful to keep accurate clocks, you can't.

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?

1.3 Serialization with messages

One solution is to instruct Bob not to eat lunch until you call. Then, make sure you don't call until after lunch. This approach may seem trivial, but the underlying idea, message passing, is a real solution for many synchroniza tion problems. At the risk of belaboring the obvious, consider this timeline.



The first column is a list of actions you perform; in other words, your thread of execution. The second column is Bob's thread of execution. Within a thread, we can always tell what order things happen. We can denote the order of events

$$a1 < a2 < a3 < a4$$

$$b1 < b2 < b3$$

where the relation a1 < a2 means that a1 happened before a2. In general, though, there is no way to compare events from different threads; for example, we have no idea who ate breakfast first (is a1 < b1?). But with message passing (the phone call) we can tell who ate lunch first (a3 < b3). Assuming that Bob has no other friends, he won't get a call until you call, so b2 > a4 . Combining all the relations, we get

b3 > b2 > a4 > a3

which proves that you had lunch before Bob.

In this case, we would say that you and Bob ate lunch sequentially, because we know the order of events, and you ate breakfast concurrently, because we don't.

When we talk about concurrent events, it is tempting to say that they happen at the same time, or simultaneously. As a shorthand, that's fine, as long as you remember the strict definition:

> Two events are concurrent if we cannot tell by looking at the program which will happen first.

Sometimes we can tell, after the program runs, which happened first, but often not, and even if we can, there is no guarantee that we will get the same result the next time.

1.4 Non-determinism

Concurrent programs are often non-deterministic, which means it is not pos sible to tell, by looking at the program, what will happen when it executes. Here is a simple example of a non-deterministic program:

Thread B

```
print " no "
```

Thread A

```
print " yes "
```

[1]

[1] Because the two threads run concurrently, the order of execution depends on the scheduler. During any given run of this program, the output might be "yes no" or "no yes".

Non-determinism is one of the things that makes concurrent programs hard to debug. A program might work correctly 1000 times in a row, and then crash on the 1001st run, depending on the particular decisions of the scheduler.

These kinds of bugs are almost impossible to find by testing; they can only be avoided by careful programming.

1.5 Shared variables

Most of the time, most variables in most threads are local, meaning that they

belong to a single thread and no other threads can access them. As long as that's true, there tend to be few synchronization problems, because threads just don't interact.

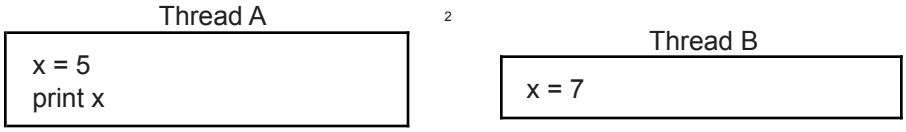
But usually some variables are shared among two or more threads; this is one of the ways threads interact with each other. For example, one way to communicate information between threads is for one thread to read a value written by another thread.

If the threads are unsynchronized, then we cannot tell by looking at the program whether the reader will see the value the writer writes or an old value that was already there. Thus many applications enforce the constraint that the reader should not read until after the writer writes. This is exactly the serialization problem in Section 1.3.

Other ways that threads interact are concurrent writes (two or more writers) and concurrent updates (two or more threads performing a read followed by a write). The next two sections deal with these interactions. The other possible use of a shared variable, concurrent reads, does not generally create a synchronization problem.

1.5.1 Concurrent writes

In the following example, x is a shared variable accessed by two writers.



What value of x gets printed? What is the final value of x when all these statements have executed? It depends on the order in which the statements are executed, called the execution path. One possible path is a1 < a2 < b1, in which case the output of the program is 5, but the final value is 7.

Puzzle: What path yields output 5 and final value 5?

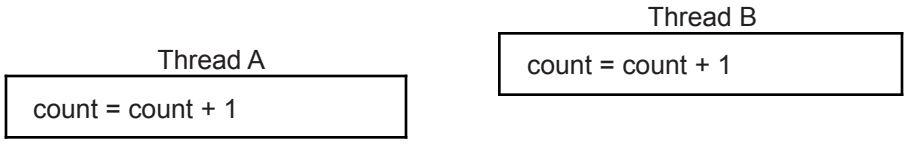
Puzzle: What path yields output 7 and final value 7?

Puzzle: Is there a path that yields output 7 and final value 5? Can you prove it?

Answering questions like these is an important part of concurrent programming: What paths are possible and what are the possible effects? Can we prove that a given (desirable) effect is necessary or that an (undesirable) effect is impossible?

1.5.2 Concurrent updates

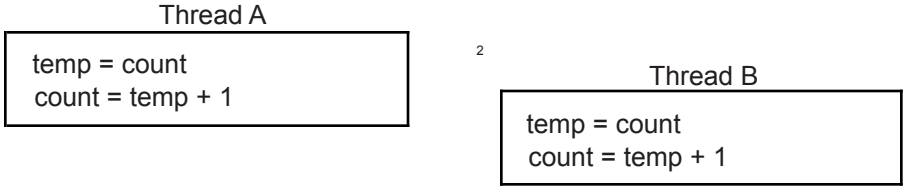
An update is an operation that reads the value of a variable, computes a new value based on the old value, and writes the new value. The most common kind of update is an increment, in which the new value is the old value plus one. The following example shows a shared variable, count, being updated concurrently by two threads.



1

1

At first glance, it is not obvious that there is a synchronization problem here. There are only two execution paths, and they yield the same result. The problem is that these operations are translated into machine language before execution, and in machine language the update takes two steps, a read and a write. The problem is more obvious if we rewrite the code with a tempo rary variable, temp.



2

1

1

2

Now consider the following execution path

a1 < b1 < b2 < a2

Assuming that the initial value of x is 0, what is its final value? Because both threads read the same initial value, they write the same value. The variable

is only incremented once, which is probably not what the programmer had in mind.

This kind of problem is subtle because it is not always possible to tell, looking at a high-level program, which operations are performed in a single step and which can be interrupted. In fact, some computers provide an increment instruction that is implemented in hardware and cannot be

interrupted. An operation that cannot be interrupted is said to be atomic.

So how can we write concurrent programs if we don't know which operations are atomic? One possibility is to collect specific information about each opera tion on each hardware platform. The drawbacks of this approach are obvious.

The most common alternative is to make the conservative assumption that all updates and all writes are not atomic, and to use synchronization constraints to control concurrent access to shared variables.

The most common constraint is mutual exclusion, or mutex, which I men tioned in Section 1.1. Mutual exclusion guarantees that only one thread accesses a shared variable at a time, eliminating the kinds of synchronization errors in this section.

Puzzle: Suppose that 100 threads run the following program concurrently (if you are not familiar with Python, the for loop runs the update 100 times.):

```
for i in range (100):
    temp = count
    count = temp + 1
```

1

2

3

What is the largest possible value of count after all threads have completed? What is the smallest possible value?

Hint: the first question is easy; the second is not.

1.5.3 Mutual exclusion with messages

Like serialization, mutual exclusion can be implemented using message passing. For example, imagine that you and Bob operate a nuclear reactor that you monitor from remote stations. Most of the time, both of you are watching for warning lights, but you are both allowed to take a break for lunch. It doesn't matter who eats lunch first, but it is very important that you don't eat lunch at the same time, leaving the reactor unwatched!

Puzzle: Figure out a system of message passing (phone calls) that enforces these restraints. Assume there are no clocks, and you cannot predict when lunch will start or how long it will last. What is the minimum number of messages that is required?

Chapter 2

Semaphores

In real life a semaphore is a system of signals used to communicate visually, usually with flags, lights, or some other mechanism. In software, a semaphore is a data structure that is useful for solving a variety of synchronization problems.

Semaphores were invented by Edsger Dijkstra, a famously eccentric com puter scientist. Some of the details have changed since the original design, but the basic idea is the same.

2.1 Definition

A semaphore is like an integer, with three differences:

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.

2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.

3. When a thread increments the semaphore, if there are other threads wait ing, one of the waiting threads gets unblocked.

To say that a thread blocks itself (or simply “blocks”) is to say that it notifies the scheduler that it cannot proceed. The scheduler will prevent the thread from running until an event occurs that causes the thread to become unblocked. In the tradition of mixed metaphors in computer science, unblocking is often called “waking”.

That’s all there is to the definition, but there are some consequences of the definition you might want to think about.

- In general, there is no way to know before a thread decrements a semaphore whether it will block or not (in specific cases you might be able to prove that it will or will not).
- After a thread increments a semaphore and another thread gets woken up, both threads continue running concurrently. There is no way to know which thread, if either, will continue immediately.
- When you signal a semaphore, you don’t necessarily know whether another thread is waiting, so the number of unblocked threads may be zero or one.

Finally, you might want to think about what the value of the semaphore means. If the value is positive, then it represents the number of threads that can decrement without blocking. If it is negative, then it represents the number of threads that have blocked and are waiting. If the value is zero, it means there are no threads waiting, but if a thread tries to decrement, it will block.

2.2 Syntax

In most programming environments, an implementation of semaphores is avail able as part of the programming language or the operating system. Different implementations sometimes offer slightly different capabilities, and usually re quire different syntax.

In this book I will use a simple pseudo-language to demonstrate how semaphores work. The syntax for creating a new semaphore and initializing it is

Semaphore initialization syntax

```
fred = Semaphore (1)
```

1

The function Semaphore is a constructor; it creates and returns a new Semaphore. The initial value of the semaphore is passed as a parameter to the constructor.

The semaphore operations go by different names in different environments. The most common alternatives are

Semaphore operations

```
fred . increment ()
fred . decrement ()
```

1

2

and

Semaphore operations

```
fred . signal ()
fred . wait ()
```

1

2

and
2.3 Why semaphores? 9

Semaphore operations

fred .V () fred .P ()

1

2

It may be surprising that there are so many names, but there is a reason. increment and decrement describe what the operations do. signal and wait describe what they are often used for. And V and P were the original names proposed by Dijkstra, who wisely realized that a meaningless name is better than a misleading name[1].

I consider the other pairs misleading because increment and decrement neglect to mention the possibility of blocking and waking, and semaphores are often used in ways that have nothing to do with signal and wait. If you insist on meaningful names, then I would suggest these:

Semaphore operations

fred . increment_and_wake_a_waiting_process_if_any () fred . decrement_and_block_if_the_result_is_negative ()

1

2

I don't think the world is likely to embrace either of these names soon. In the meantime, I choose (more or less arbitrarily) to use signal and wait.

2.3 Why semaphores?

Looking at the definition of semaphores, it is not at all obvious why they are use ful. It's true that we don't need semaphores to solve synchronization problems, but there are some advantages to using them:

- Semaphores impose deliberate constraints that help programmers avoid errors.
- Solutions using semaphores are often clean and organized, making it easy to demonstrate their correctness.

• Semaphores can be implemented efficiently on many systems, so solutions that use semaphores are portable and usually efficient.

[1] If you speak Dutch, V and P aren't completely meaningless.

Chapter 3

Basic synchronization patterns

This chapter presents a series of basic synchronization problems and shows ways of using semaphores to solve them. These problems include serialization and mutual exclusion, which we have already seen, along with others.

3.1 Signaling

Possibly the simplest use for a semaphore is signaling, which means that one thread sends a signal to another thread to indicate that something has happened. Signaling makes it possible to guarantee that a section of code in one thread will run before a section of code in another thread; in other words, it solves the serialization problem.

Assume that we have a semaphore named sem with initial value 0, and that Threads A and B have shared access to it.

Thread A

```
statement a1
sem . signal ()
```

1

1

2

```
sem . wait ()
statement b1
```

Thread B

The word statement represents an arbitrary program statement. To make the example concrete, imagine that a1 reads a line from a file, and b1 displays the line on the screen. The semaphore in this program guarantees that Thread A has completed a1 before Thread B begins b1.

Here's how it works: if thread B gets to the wait statement first, it will find the initial value, zero, and it will block. Then when Thread A signals, Thread B proceeds.

Similarly, if Thread A gets to the signal first then the value of the semaphore will be incremented, and when Thread B gets to the wait, it will proceed im-

mediately. Either way, the order of a1 and b1 is guaranteed. This use of semaphores is the basis of the names signal and wait, and in this case the names are conveniently mnemonic. Unfortunately, we will see other cases where the names are less helpful.

Speaking of meaningful names, sem isn't one. When possible, it is a good idea to give a semaphore a name that indicates what it represents. In this case a name like a1Done might be good, so that a1done.signal() means "signal that a1 is done," and a1done.wait() means "wait until a1 is done."

3.2 Sync.py

TODO: write about using sync, starting with signal.py

Why does Thread B signal initComplete?

3.3 Rendezvous

Puzzle: Generalize the signal pattern so that it works both ways. Thread A has to wait for Thread B and vice versa. In other words, given this code

Thread A

```
statement a1
statement a2
```

Thread B

```
statement b1
statement b2
```

we want to guarantee that a1 happens before b2 and b1 happens before a2. In writing your solution, be sure to specify the names and initial values of your semaphores (little hint there).

Your solution should not enforce too many constraints. For example, we don't care about the order of a1 and b1. In your solution, either order should be possible.

This synchronization problem has a name; it's a rendezvous. The idea is that two threads rendezvous at a point of execution, and neither is allowed to proceed until both have arrived.

3.3.1 Rendezvous hint

The chances are good that you were able to figure out a solution, but if not, here is a hint. Create two semaphores, named aArrived and bArrived, and initialize them both to zero.

As the names suggest, aArrived indicates whether Thread A has arrived at the rendezvous, and bArrived likewise.

3.3.2 Rendezvous solution

Here is my solution, based on the previous hint:

Thread A

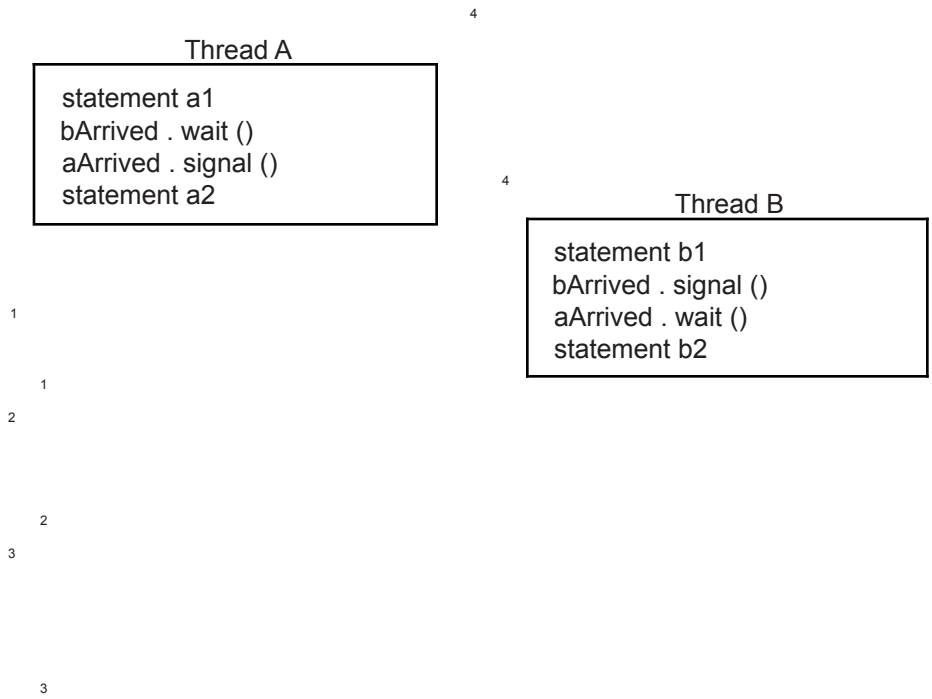
```
statement a1
aArrived . signal ()
bArrived . wait ()
statement a2
```

Thread B

```
statement b1
bArrived . signal ()
aArrived . wait ()
statement b2
```

While working on the previous problem, you might have tried something

like this:

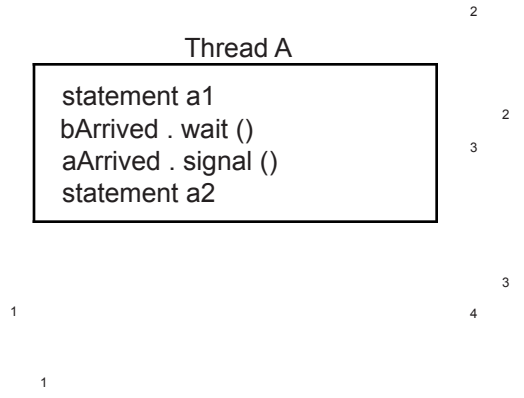


This solution also works, although it is probably less efficient, since it might have to switch between A and B one time more than necessary. If A arrives first, it waits for B. When B arrives, it wakes A and might proceed immediately to its wait in which case it blocks, allowing A to reach its signal, after which both threads can proceed.

Think about the other possible paths through this code and convince yourself that in all cases neither thread can proceed until both have arrived.

3.3.3 Deadlock #1

Again, while working on the previous problem, you might have tried something like this:



```
statement b1
aArrived . wait ()
bArrived . signal ()
statement b2
```

4

Thread B

If so, I hope you rejected it quickly, because it has a serious problem. As suming that A arrives first, it will block at its wait. When B arrives, it will also block, since A wasn't able to signal aArrived. At this point, neither thread can proceed, and never will.

This situation is called a deadlock and, obviously, it is not a successful solution of the synchronization problem. In this case, the error is obvious, but often the possibility of deadlock is more subtle. We will see more examples later.

3.4 Mutex

A second common use for semaphores is to enforce mutual exclusion. We have al ready seen one use for mutual exclusion, controlling concurrent access to shared variables. The mutex guarantees that only one thread accesses the shared vari able at a time.

A mutex is like a token that passes from one thread to another, allowing one thread at a time to proceed. For example, in The Lord of the Flies a group of children use a conch as a mutex. In order to speak, you have to hold the conch. As long as only one child holds the conch, only one can speak[1].

Similarly, in order for a thread to access a shared variable, it has to "get" the mutex; when it is done, it "releases" the mutex. Only one thread can hold the mutex at a time.

Puzzle: Add semaphores to the following example to enforce mutual exclu sion to the shared variable count.

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

1

1

[1]Although this metaphor is helpful for now, it can also be misleading, as you will see in Section 5.6

3.4.1 Mutual exclusion hint

Create a semaphore named mutex that is initialized to 1. A value of one means that a thread may proceed and access the shared variable; a value of zero means that it has to wait for another thread to release the mutex.

3.4.2 Mutual exclusion solution

Here is a solution:

Thread A

```
mutex . wait ()
        # critical section
    count = count + 1
mutex . signal ()
```

Thread B

```
mutex . wait ()
        # critical section
    count = count + 1
mutex . signal ()
```

Since mutex is initially 1, whichever thread gets to the wait first will be able to proceed immediately. Of course, the act of waiting on the semaphore has the effect of decrementing it, so the second thread to arrive will have to wait until the first signals.

I have indented the update operation to show that it is contained within the mutex.

In this example, both threads are running the same code. This is sometimes called a symmetric solution. If the threads have to run different code, the solu tion is asymmetric. Symmetric solutions are often easier to generalize. In this case, the mutex solution can handle any number of concurrent threads without modification. As long as every thread waits before performing an update and signals after, then no two threads will access count concurrently.

Often the code that needs to be protected is called the critical section, I suppose because it is critically important to prevent concurrent access. In the tradition of computer science and mixed metaphors, there are several other

ways people sometimes talk about mutexes. In the metaphor we have been using so far, the mutex is a token that is passed from one thread to another. In an alternative metaphor, we think of the critical section as a room, and only one thread is allowed to be in the room at a time. In this metaphor, mutexes are called locks, and a thread is said to lock the mutex before entering and unlock it while exiting. Occasionally, though, people mix the metaphors and talk about “getting” or “releasing” a lock, which doesn’t make much sense. Both metaphors are potentially useful and potentially misleading. As you work on the next problem, try out both ways of thinking and see which one leads you to a solution.

3.5 Multiplex

Puzzle: Generalize the previous solution so that it allows multiple threads to run in the critical section at the same time, but it enforces an upper limit on the number of concurrent threads. In other words, no more than n threads can run in the critical section at the same time.

This pattern is called a multiplex. In real life, the multiplex problem occurs at busy nightclubs where there is a maximum number of people allowed in the

building at a time, either to maintain fire safety or to create the illusion of exclusivity.

At such places a bouncer usually enforces the synchronization constraint by keeping track of the number of people inside and barring arrivals when the room is at capacity. Then, whenever one person leaves another is allowed to enter.

Enforcing this constraint with semaphores may sound difficult, but it is almost trivial.

3.5.1 Multiplex solution

To allow multiple threads to run in the critical section, just initialize the semaphore to n, which is the maximum number of threads that should be al lowed.

At any time, the value of the semaphore represents the number of additional threads that may enter. If the value is zero, then the next thread will block until one of the threads inside exits and signals. When all threads have exited the value of the semaphore is restored to n.

Since the solution is symmetric, it’s conventional to show only one copy of the code, but you should imagine multiple copies of the code running concurrently in multiple threads.

Multiplex solution

```
multiplex . wait ()
        critical section
multiplex . signal ()
```

1
2
3

What happens if the critical section is occupied and more than one thread arrives? Of course, what we want is for all the arrivals to wait. This solution does exactly that. Each time an arrival joins the queue, the semaphore is decre mented, so that the value of the semaphore (negated) represents the number of threads in queue.

When a thread leaves, it signals the semaphore, incrementing its value and allowing one of the waiting threads to proceed.

Thinking again of metaphors, in this case I find it useful to think of the semaphore as a set of tokens (rather than a lock). As each thread invokes wait, it picks up one of the tokens; when it invokes signal it releases one. Only a thread that holds a token can enter the room. If no tokens are available when a thread arrives, it waits until another thread releases one.

In real life, ticket windows sometimes use a system like this. They hand out tokens (sometimes poker chips) to customers in line. Each token allows the holder to buy a ticket.

3.6 Barrier

Consider again the Rendezvous problem from Section 3.3. A limitation of the solution we presented is that it does not work with more than two threads. Puzzle: Generalize the rendezvous solution. Every thread should run the following code:

Barrier code

```
rendezvous
critical point
```

1
2

The synchronization requirement is that no thread executes critical point until after all threads have executed rendezvous.

You can assume that there are n threads and that this value is stored in a

variable, n, that is accessible from all threads.

When the first n − 1 threads arrive they should block until the nth thread arrives, at which point all the threads may proceed.

3.6 Barrier 23

3.6.1 Barrier hint

For many of the problems in this book I will provide hints by presenting the variables I used in my solution and explaining their roles.

Barrier hint

```
n = the number of threads
count = 0
mutex = Semaphore (1)
barrier = Semaphore (0)
```

1

2

3

4

count keeps track of how many threads have arrived. mutex provides exclu sive access to count so that threads can increment it safely.

barrier is locked (zero or negative) until all threads arrive; then it should be unlocked (1 or more).

3.6 Barrier 25

3.6.2 Barrier non-solution

First I will present a solution that is not quite right, because it is useful to examine incorrect solutions and figure out what is wrong.

Barrier non-solution

```
rendezvous

mutex . wait ()
        count = count + 1
mutex . signal ()

if count == n: barrier . signal ()

barrier . wait ()

critical point
```

Since count is protected by a mutex, it counts the number of threads that pass. The first n−1 threads wait when they get to the barrier, which is initially locked. When the nth thread arrives, it unlocks the barrier.

Puzzle: What is wrong with this solution?

26 Basic synchronization patterns

3.6 Barrier 27

3.6.3 Deadlock #2

The problem is a deadlock.

An an example, imagine that n = 5 and that 4 threads are waiting at the barrier. The value of the semaphore is the number of threads in queue, negated, which is -4.

When the 5th thread signals the barrier, one of the waiting threads is allowed to proceed, and the semaphore is incremented to -3.

But then no one signals the semaphore again and none of the other threads can pass the barrier. This is a second example of a deadlock. Puzzle: Does this code always create a deadlock? Can you find an execution path through this code that does not cause a deadlock?

Puzzle: Fix the problem.

28 Basic synchronization patterns

3.6 Barrier 29

3.6.4 Barrier solution

Finally, here is a working barrier:

Barrier solution

```
rendezvous

mutex . wait ()
        count = count + 1
mutex . signal ()

if count == n: barrier . signal ()

barrier . wait ()
barrier . signal ()

critical point
```

1
2
3
4
5
6
7
8
9
10
11
12

The only change is another signal after waiting at the barrier. Now as each thread passes, it signals the semaphore so that the next thread can pass. This pattern, a wait and a signal in rapid succession, occurs often enough that it has a name; it's called a turnstile, because it allows one thread to pass at a time, and it can be locked to bar all threads.

In its initial state (zero), the turnstile is locked. The nth thread unlocks it and then all n threads go through.

It might seem dangerous to read the value of count outside the mutex. In this case it is not a problem, but in general it is probably not a good idea. We will clean this up in a few pages, but in the meantime, you might want to consider these questions: After the nth thread, what state is the turnstile in? Is there any way the barrier might be signaled more than once?

3.6.5 Deadlock #3

Since only one thread at a time can pass through the mutex, and only one thread at a time can pass through the turnstile, it might seen reasonable to put the turnstile inside the mutex, like this:

Bad barrier solution

```
rendezvous

mutex . wait ()
        count = count + 1
        if count == n: barrier . signal ()

        barrier . wait ()
        barrier . signal ()
mutex . signal ()

critical point
```

1
2
3
4
5
6
7
8
9
10
11

This turns out to be a bad idea because it can cause a deadlock. Imagine that the first thread enters the mutex and then blocks when it reaches the turnstile. Since the mutex is locked, no other threads can enter, so the condition, count==n, will never be true and no one will ever unlock the turnstile.

In this case the deadlock is fairly obvious, but it demonstrates a common source of deadlocks: blocking on a semaphore while holding a mutex.

3.7 Reusable barrier

Often a set of cooperating threads will perform a series of steps in a loop and synchronize at a barrier after each step. For this application we need a reusable barrier that locks itself after all the threads have passed through.

Puzzle: Rewrite the barrier solution so that after all the threads have passed through, the turnstile is locked again.

3.7.1 Reusable barrier non-solution #1

Once again, we will start with a simple attempt at a solution and gradually improve it:

Reusable barrier non-solution

```
rendezvous

mutex . wait ()
        count += 1
mutex . signal ()

if count == n: turnstile . signal ()

turnstile . wait ()
turnstile . signal ()

critical point

mutex . wait ()
        count -= 1
mutex . signal ()

if count == 0: turnstile . wait ()
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

Notice that the code after the turnstile is pretty much the same as the code before it. Again, we have to use the mutex to protect access to the shared variable count. Tragically, though, this code is not quite correct. Puzzle: What is the problem?

3.7.2 Reusable barrier problem #1

There is a problem spot at Line 7 of the previous code.

If the n − 1th thread is interrupted at this point, and then the nth thread comes through the mutex, both threads will find that count==n and both threads will signal the turnstile. In fact, it is even possible that all the threads will signal the turnstile.

Similarly, at Line 18 it is possible for multiple threads to wait, which will cause a deadlock.

Puzzle: Fix the problem.

3.7.3 Reusable barrier non-solution #2

This attempt fixes the previous error, but a subtle problem remains.

Reusable barrier non-solution

```
rendezvous

mutex . wait ()
        count += 1
        if count == n: turnstile . signal ()
mutex . signal ()

turnstile . wait ()
turnstile . signal ()

critical point

mutex . wait ()
        count -= 1
        if count == 0: turnstile . wait ()
mutex . signal ()
```

1
2
3
4
5
6
7
8

9
10
11
12
13
14
15
16

In both cases the check is inside the mutex so that a thread cannot be interrupted after changing the counter and before checking it.

Tragically, this code is still not correct. Remember that this barrier will be inside a loop. So, after executing the last line, each thread will go back to the rendezvous.

Puzzle: Identify and fix the problem.

3.7.4 Reusable barrier hint

As it is currently written, this code allows a precocious thread to pass through the second mutex, then loop around and pass through the first mutex and the turnstile, effectively getting ahead of the other threads by a lap. To solve this problem we can use two turnstiles.

Reusable barrier hint

```
turnstile = Semaphore (0)
turnstile2 = Semaphore (1)
mutex = Semaphore (1)
```

1
2
3

Initially the first is locked and the second is open. When all the threads arrive at the first, we lock the second and unlock the first. When all the threads arrive at the second we relock the first, which makes it safe for the threads to loop around to the beginning, and then open the second.

 3.7.5 Reusable barrier solution

Reusable barrier solution

```
# rendezvous

mutex . wait ()
        count += 1
        if count == n:
                                        turnstile2 . wait () # lock the second
                                          turnstile . signal () # unlock the first
mutex . signal ()

turnstile . wait () # first turnstile turnstile . signal ()

# critical point

mutex . wait ()
        count -= 1
        if count == 0:
                                                turnstile . wait () # lock the first
                turnstile2 . signal () # unlock the second mutex . signal ()

turnstile2 . wait () # second turnstile turnstile2 . signal ()
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22

This solution is sometimes called a two-phase barrier because it forces all the threads to wait twice: once for all the threads to arrive and again for all the threads to execute the critical section.

Unfortunately, this solution is typical of most non-trivial synchronization code: it is difficult to be sure that a solution is correct. Often there is a subtle way that a particular path through the program can cause an error.

To make matters worse, testing an implementation of a solution is not much help. The error might occur very rarely because the particular path that causes it might require a spectacularly unlucky combination of circumstances. Such errors are almost impossible to reproduce and debug by conventional means.

The only alternative is to examine the code carefully and “prove” that it is correct. I put “prove” in quotation marks because I don’t mean, necessarily, that you have to write a formal proof (although there are zealots who encourage such lunacy).

The kind of proof I have in mind is more informal. We can take advantage of the structure of the code, and the idioms we have developed, to assert, and then demonstrate, a number of intermediate-level claims about the program. For example:

1. Only the nth thread can lock or unlock the turnstiles.

2. Before a thread can unlock the first turnstile, it has to close the second, and vice versa; therefore it is impossible for one thread to get ahead of the others by more than one turnstile.

By finding the right kinds of statements to assert and prove, you can sometimes find a concise way to convince yourself (or a skeptical colleague) that your code is bulletproof.

3.7.6 Preloaded turnstile

One nice thing about a turnstile is that it is a versatile component you can use in a variety of solutions. But one drawback is that it forces threads to go through sequentially, which may cause more context switching than necessary.

In the reusable barrier solution, we can simplify the solution if the thread that unlocks the turnstile preloads the turnstile with enough signals to let the right number of threads through[2].

The syntax I am using here assumes that signal can take a parameter that specifies the number of signals. This is a non-standard feature, but it would be easy to implement with a loop. The only thing to keep in mind is that the multiple signals are not atomic; that is, the signaling thread might be

interrupted in the loop. But in this case that is not a problem.

Reusable barrier solution

```
# rendezvous

mutex . wait ()
        count += 1
        if count == n:
                turnstile . signal (n) # unlock the first mutex . signal ()

turnstile . wait () # first turnstile # critical point

mutex . wait ()
        count -= 1
        if count == 0:
                turnstile2 . signal (n ) # unlock the second mutex . signal ()

turnstile2 . wait () # second turnstile
```

When the nth thread arrives, it preloads the first turnstile with one signal for each thread. When the nth thread passes the turnstile, it “takes the last token” and leaves the turnstile locked again.

The same thing happens at the second turnstile, which is unlocked when

the last thread goes through the mutex.

[2]Thanks to Matt Tesch for this solution!

3.7.7 Barrier objects

It is natural to encapsulate a barrier in an object. I will borrow the Python syntax for defining a class:

Barrier class

```
class Barrier :
        def __init__ ( self , n ):
                self .n = n
                self . count = 0
                self . mutex = Semaphore (1)
                self . turnstile = Semaphore (0)
                self . turnstile2 = Semaphore (0)

        def phase1 ( self ):
                self . mutex . wait ()
                        self . count += 1
                        if self . count == self .n:
                                        self . turnstile . signal ( self . n)
                self . mutex . signal ()
                self . turnstile . wait ()

        def phase2 ( self ):
                self . mutex . wait ()
                        self . count -= 1
                        if self . count == 0:
                                         self . turnstile2 . signal ( self .n)
                self . mutex . signal ()
                self . turnstile2 . wait ()

        def wait ( self ):
                self . phase1 ()
                self . phase2 ()
```

1

2

The init method runs when we create a new Barrier object, and ini tializes the instance variables. The parameter n is the number of threads that have to invoke wait before the Barrier opens.

The variable self refers to the object the method is operating on. Since each barrier object has its own mutex and turnstiles, self.mutex refers to the specific mutex of the current object.

Here is an example that creates a Barrier object and waits on it:

Barrier interface

```
barrier = Barrier (n) # initialize a new barrier barrier . wait () # wait at a barrier
```


Optionally, code that uses a barrier can call phase1 and phase2 separately, if there is something else that should be done in between.

3.8 Queue

Semaphores can also be used to represent a queue. In this case, the initial value is 0, and usually the code is written so that it is not possible to signal unless there is a thread waiting, so the value of the semaphore is never positive.

For example, imagine that threads represent ballroom dancers and that two kinds of dancers, leaders and followers, wait in two queues before entering the dance floor. When a leader arrives, it checks to see if there is a follower waiting. If so, they can both proceed. Otherwise it waits.

Similarly, when a follower arrives, it checks for a leader and either proceeds or waits, accordingly.

Puzzle: write code for leaders and followers that enforces these constraints.

3.8.1 Queue hint

Here are the variables I used in my solution:

Queue hint

```
leaderQueue = Semaphore (0)
followerQueue = Semaphore (0)
```

1

2

leaderQueue is the queue where leaders wait and followerQueue is the queue where followers wait.

3.8.2 Queue solution

Here is the code for leaders:

Queue solution (leaders)

```
followerQueue . signal ()
leaderQueue . wait ()
dance ()
```

1

2
3

And here is the code for followers:

Queue solution (followers)

```
leaderQueue . signal ()
followerQueue . wait ()
dance ()
```

1
2
3

This solution is about as simple as it gets; it is just a Rendezvous. Each leader signals exactly one follower, and each follower signals one leader, so it is guaranteed that leaders and followers are allowed to proceed in pairs. But whether they actually proceed in pairs is not clear. It is possible for any number of threads to accumulate before executing dance, and so it is possible for any number of leaders to dance before any followers do. Depending on the semantics of dance, that behavior may or may not be problematic.

To make things more interesting, let's add the additional constraint that each leader can invoke dance concurrently with only one follower, and vice versa. In other words, you got to dance with the one that brought you[3].

Puzzle: write a solution to this "exclusive queue" problem.

[3]Song lyric performed by Shania Twain

3.8.3 Exclusive queue hint

Here are the variables I used in my solution:

Queue hint

```
leaders = followers = 0
mutex = Semaphore (1)
leaderQueue = Semaphore (0)
followerQueue = Semaphore (0)
rendezvous = Semaphore (0)
```

1
2
3
4
5

leaders and followers are counters that keep track of the number of dancers of each kinds that are waiting. The mutex guarantees exclusive ac cess to the counters.

leaderQueue and followerQueue are the queues where dancers wait. rendezvous is used to check that both threads are done dancing.

3.8.4 Exclusive queue solution

Here is the code for leaders:

Queue solution (leaders)

```
mutex . wait ()
if followers > 0:
        followers --
        followerQueue . signal ()
else :
        leaders ++
        mutex . signal ()
        leaderQueue . wait ()

dance ()
rendezvous . wait ()
mutex . signal ()
```

When a leader arrives, it gets the mutex that protects leaders and followers. If there is a follower waiting, the leader decrements followers, signals a follower, and then invokes dance, all before releasing mutex. That guarantees that there can be only one follower thread running dance concur rently.

If there are no followers waiting, the leader has to give up the mutex before waiting on leaderQueue.

The code for followers is similar:

Queue solution (followers)

```
mutex . wait ()
if leaders > 0:
        leaders --
        leaderQueue . signal ()
else :
        followers ++
        mutex . signal ()
        followerQueue . wait ()

dance ()
rendezvous . signal ()
```

When a follower arrives, it checks for a waiting leader. If there is one, the follower decrements leaders, signals a leader, and executes dance, all without releasing mutex. Actually, in this case the follower never releases mutex; the leader does. We don't have to keep track of which thread has the mutex because we know that one of them does, and either one of them can release it. In my solution it's always the leader.

When a semaphore is used as a queue[4], I find it useful to read "wait" as "wait for this queue" and signal as "let someone from this queue go." In this code we never signal a queue unless someone is waiting, so the values of the queue semaphores are seldom positive. It is possible, though. See if you can figure out how.

[4]A semaphore used as a queue is very similar to a condition variable. The primary difference is that threads have to release the mutex explicitly before waiting, and reacquire it explicitly afterwards (but only if they need it).

Chapter 4

Classical synchronization

problems

In this chapter we examine the classical problems that appear in nearly every operating systems textbook. They are usually presented in terms of real-world problems, so that the statement of the problem is clear and so that students can bring their intuition to bear.

For the most part, though, these problems do not happen in the real world, or if they do, the real-world solutions are not much like the kind of synchronization code we are working with.

The reason we are interested in these problems is that they are analogous to common problems that operating systems (and some applications) need to solve. For each classical problem I will present the classical formulation, and also explain the analogy to the corresponding OS problem.

4.1 Producer-consumer problem

In multithreaded programs there is often a division of labor between threads. In one common pattern, some threads are producers and some are consumers. Pro ducers create items of some kind and add them to a data structure; consumers remove the items and process them.

Event-driven programs are a good example. An “event” is something that happens that requires the program to respond: the user presses a key or moves the mouse, a block of data arrives from the disk, a packet arrives from the network, a pending operation completes.

Whenever an event occurs, a producer thread creates an event object and adds it to the event buffer. Concurrently, consumer threads take events out of the buffer and process them. In this case, the consumers are called “event handlers.”

There are several synchronization constraints that we need to enforce to make this system work correctly:

- While an item is being added to or removed from the buffer, the buffer is in an inconsistent state. Therefore, threads must have exclusive access to the buffer.
- If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item.

Assume that producers perform the following operations over and over:

Basic producer code

```
event = waitForEvent ()
buffer . add ( event )
```

1

2

Also, assume that consumers perform the following operations:

Basic consumer code

```
event = buffer . get ()
event . process ()
```

1

2

As specified above, access to the buffer has to be exclusive, but waitForEvent and event.process can run concurrently.

Puzzle: Add synchronization statements to the producer and consumer code to enforce the synchronization constraints.

4.1.1 Producer-consumer hint

Here are the variables you might want to use:

Producer-consumer initialization

```
mutex = Semaphore (1)
items = Semaphore (0)
local event
```

1

2

3

Not surprisingly, mutex provides exclusive access to the buffer. When items is positive, it indicates the number of items in the buffer. When it is negative, it indicates the number of consumer threads in queue.

event is a local variable, which in this context means that each thread has its own version. So far we have been assuming that all threads have access to all variables, but we will sometimes find it useful to attach a variable to each thread.

There are a number of ways this can be implemented in different environments:

- If each thread has its own run-time stack, then any variables allocated on the stack are thread-specific.
- If threads are represented as objects, we can add an attribute to each thread object.
- If threads have unique IDs, we can use the IDs as an index into an array or hash table, and store per-thread data there.

In most programs, most variables are local unless declared otherwise, but in this book most variables are shared, so we will assume that that variables are shared unless they are explicitly declared local.

4.1.2 Producer-consumer solution

Here is the producer code from my solution.

Producer solution

```
event = waitForEvent ()
mutex . wait ()
        buffer . add ( event )
        items . signal ()
mutex . signal ()
```

The producer doesn't have to get exclusive access to the buffer until it gets an event. Several threads can run waitForEvent concurrently. The items semaphore keeps track of the number of items in the buffer. Each time the producer adds an item, it signals items, incrementing it by one. The consumer code is similar.

Consumer solution

```
items . wait ()
mutex . wait ()
        event = buffer . get ()
mutex . signal ()
event . process ()
```

Again, the buffer operation is protected by a mutex, but before the consumer gets to it, it has to decrement items. If items is zero or negative, the consumer blocks until a producer signals.

Although this solution is correct, there is an opportunity to make one small improvement to its performance. Imagine that there is at least one consumer in queue when a producer signals items. If the scheduler allows the consumer to run, what happens next? It immediately blocks on the mutex that is (still) held by the producer.

Blocking and waking up are moderately expensive operations; performing them unnecessarily can impair the performance of a program. So it would probably be better to rearrange the producer like this:

Improved producer solution

```
event = waitForEvent ()
mutex . wait ()
        buffer . add ( event )
mutex . signal ()
items . signal ()
```

Now we don't bother unblocking a consumer until we know it can proceed (except in the rare case that another producer beats it to the mutex). There's one other thing about this solution that might bother a stickler. In the hint section I claimed that the items semaphore keeps track of the number

of items in queue. But looking at the consumer code, we see the possibility that several consumers could decrement items before any of them gets the mutex and removes an item from the buffer. At least for a little while, items would be inaccurate.

We might try to address that by checking the buffer inside the mutex:

Broken consumer solution

```
mutex . wait ()
        items . wait ()
        event = buffer . get ()
mutex . signal ()
event . process ()
```

1
2
3
4
5

This is a bad idea.
Puzzle: why?

4.1.3 Deadlock #4

If the consumer is running this code

Broken consumer solution

```
mutex . wait ()
        items . wait ()
        event = buffer . get ()
mutex . signal ()

event . process ()
```

1
2
3
4
5
6

it can cause a deadlock. Imagine that the buffer is empty. A consumer arrives, gets the mutex, and then blocks on items. When the producer arrives, it blocks on mutex and the system comes to a grinding halt.

This is a common error in synchronization code: any time you wait for a semaphore while holding a mutex, there is a danger of deadlock. When you are checking a solution to a synchronization problem, you should check for this kind of deadlock.

4.1.4 Producer-consumer with a finite buffer

In the example I described above, event-handling threads, the shared buffer is usually infinite (more accurately, it is bounded by system resources like physical memory and swap space).

In the kernel of the operating system, though, there are limits on available space. Buffers for things like disk requests and network packets are usually fixed size. In situations like these, we have an additional synchronization constraint:

- If a producer arrives when the buffer is full, it blocks until a consumer removes an item.

Assume that we know the size of the buffer. Call it bufferSize. Since we have a semaphore that is keeping track of the number of items, it is tempting to write something like

Broken finite buffer solution

```
if items >= bufferSize :
        block ()
```

1
2

But we can't. Remember that we can't check the current value of a semaphore; the only operations are wait and signal.

Puzzle: write producer-consumer code that handles the finite-buffer con straint.

4.1.5 Finite buffer producer-consumer hint

Add a second semaphore to keep track of the number of available spaces in the buffer.

Finite-buffer producer-consumer initialization

```
mutex = Semaphore (1)
items = Semaphore (0)
spaces = Semaphore ( buffer . size ())
```

1
2
3

When a consumer removes an item it should signal spaces. When a

producer arrives it should decrement spaces, at which point it might block until the next consumer signals.

4.1.6 Finite buffer producer-consumer solution

Here is a solution.

Finite buffer consumer solution

```
items . wait ()
mutex . wait ()
        event = buffer . get ()
mutex . signal ()
spaces . signal ()

event . process ()
```

1
2
3
4
5
6
7

The producer code is symmetric, in a way:

Finite buffer producer solution

```
event = waitForEvent ()

spaces . wait ()
mutex . wait ()
        buffer . add ( event )
mutex . signal ()
items . signal ()
```

1
2
3
4
5
6
7

In order to avoid deadlock, producers and consumers check availability be fore getting the mutex. For best performance, they release the mutex before signaling.

4.2 Readers-writers problem

The next classical problem, called the Reader-Writer Problem, pertains to any situation where a data structure, database, or file system is read and modified by concurrent threads. While the data structure is being written or modified it is often necessary to bar other threads from reading, in order to prevent a reader from interrupting a modification in progress and reading inconsistent or invalid data.

As in the producer-consumer problem, the solution is asymmetric. Readers and writers execute different code before entering the critical section. The synchronization constraints are:

1. Any number of readers can be in the critical section simultaneously.

2. Writers must have exclusive access to the critical section.

In other words, a writer cannot enter the critical section while any other thread (reader or writer) is there, and while the writer is there, no other thread may enter.

The exclusion pattern here might be called categorical mutual exclusion. A thread in the critical section does not necessarily exclude other threads, but the presence of one category in the critical section excludes other categories.

Puzzle: Use semaphores to enforce these constraints, while allowing readers and writers to access the data structure, and avoiding the possibility of deadlock.

4.2.1 Readers-writers hint

Here is a set of variables that is sufficient to solve the problem.

Readers-writers initialization

```
int readers = 0
mutex = Semaphore (1)
roomEmpty = Semaphore (1)
```

1

2

3

The counter readers keeps track of how many readers are in the room. mutex protects the shared counter.

roomEmpty is 1 if there are no threads (readers or writers) in the critical section, and 0 otherwise. This demonstrates the naming convention I use for semaphores that indicate a condition. In this convention, “wait” usually means “wait for the condition to be true” and “signal” means “signal that the condition is true”.