The Little Book of Semaphores

Allen B. Downey

The Little Book of Semaphores
First Edition

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.

The LaTeX source for this book, and more information about the Open Source Textbook project, is available from `allendowney.com/semaphores`.

This book was typeset by the author using latex, dvips and ps2pdf, among other free, open-source programs.

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 synchonization 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?

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 synchronization problems.

At the risk of belaboring the obvious, consider this timeline.

You		Bob	
a1	`Eat breakfast`	b1	`Eat breakfast`
a2	`Work`	b2	`Wait for a call`
a3	`Eat lunch`	b3	`Eat lunch`
a4	`Call Bob`		

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 possible to tell, by looking at the program, what will happen when it executes.

Here is a very simple example of a non-deterministic program:

Thread A

```
a1  print "yes"
```

Thread B

```
b1  print "no"
```

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 contraint 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.

Thread A

```
a1  x = 5
a2  print x
```

Thread B

```
b1  x = 7
```

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.

Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 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 temporary variable, `temp`.

Thread A

```
a1  temp = count
a2  count = temp + 1
```

Thread B

```
b1  temp = count
b2  count = temp + 1
```

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 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 operation 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 mentioned 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.

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 Edsgar Dijkstra, a famously eccentric computer 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. If the value of the semaphore is negative and a thread increments it, one of the threads that is waiting gets woken up.

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 it is notified otherwise. Since the blocked thread cannot run, some other thread has to unblock it. In the tradition of mixed computer science metaphors, 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.

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 available as part of the programming language or the operating system. Different implementations sometimes offer slightly different capabilities, and usually require 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

Listing 2.1: Semaphore initialization syntax

```
Semaphore fred = 1
```

When `Semaphore` appears with a capital letter it indicates a type of variable. In this case, `fred` is the name of the new semaphore and `1` is its initial value.

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

Listing 2.2: Semaphore operations

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

and

Listing 2.3: Semaphore operations

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

and

Listing 2.4: Semaphore operations

```
fred.V ()
fred.P ()
```

It may be surprising that there are so many names, but there is a reason for the plurality. `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:

Listing 2.5: Semaphore operations

```
fred.increment_and_wake_a_waiting_process_if_any ()
fred.decrement_and_block_if_the_result_is_negative ()
```

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 useful. 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] Actually, `V` and `P` aren't completely meaningless to people who speak Dutch.

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 ()
```

Thread B

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

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 immediately. 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, where `a1Done = 0` means that `a1` has not executed and `a1Done = 1` means it has.

3.2 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.

3.2.1 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.

3.2.2 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:

Thread A

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

Thread B

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

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.

3.2.3 Deadlock #1

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

Thread A

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

Thread B

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

If so, I hope you rejected it quickly, because it has a serious problem. Assuming 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.

3.3 Mutex

A second common use for semaphores is to enforce mutual exclusion. We have already seen one use for mutual exclusion, controlling concurrent access to shared variables. The mutex guarantees that only one thread accesses the shared variable 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.

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 exclusion to the shared variable `count`.

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

3.3.1 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.

3.3.2 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 solution 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.

3.4 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.

3.4.1 Multiplex solution

To allow multiple threads to run in the critical section, just initialize the mutex to `n`, which is the maximum number of threads that should be allowed.

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.

Listing 3.1: Multiplex solution

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

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 decremented, 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.

3.5 Barrier

Consider again the Rendezvous problem from Section 3.2. 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:

Listing 3.2: Barrier code

```
rendezvous
critical point
```

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.

3.5.1 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.

Listing 3.3: Barrier hint

```
int n                // the number of threads
int count = 0
Semaphore mutex = 1
Semaphore barrier = 0
```

`count` keeps track of how many threads have arrived. `mutex` provides exclusive 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).

3.5.2 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.

Listing 3.4: 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?

3.5.3 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: Fix this problem.

3.5.4 Barrier solution

Finally, here is a working barrier:

Listing 3.5: Barrier solution

```
rendezvous

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

if (count == n) barrier.signal ()

barrier.wait ()
barrier.signal ()

critical point
```

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.

After the nth thread, what state is the turnstile in? Is there any way the barrier might be signalled more than once?

3.5.5 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:

Listing 3.6: Bad barrier solution

```
rendezvous

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

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

critical point
```

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.

Does this code always create a deadlock? Can you find an execution path through this code that does *not* cause a deadlock?

3.6 Reusable barrier

Often a set of cooperating threads will perform a series of steps in a loop and synchonize 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.

3.6.1 Reusable barrier hint

Here's a hint about how to do it. Just as the nth thread unlocked the turnstile on the way in, it should be the nth thread that locks it on the way out.

Here are the variables I used:

Listing 3.7: Reusable barrier hint

```
int count = 0
Semaphore mutex = 1
Semaphore turnstile = 0
```

3.6.2 Reusable barrier non-solution #1

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

Listing 3.8: Reusable barrier non-solution

```
rendezvous

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

if (count == n) turnstile.signal ()

turnstile.wait ()
turnstile.signal ()

critical point

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

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

`count++` is shorthand for `count = count + 1` and `count--` for `count = count - 1`

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?

3.6.3 Reusable barrier problem #1

There is a problem spot at Line 6 of the previous code.

If the $n-1$th 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.

If this barrier isn't reused, then multiple signals are not a problem. For a reusable barrier, they are.

Puzzle: Fix the problem.

3.6.4 Reusable barrier non-solution #2

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

Listing 3.9: Reusable barrier non-solution

```
rendezvous

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

turnstile.wait ()
turnstile.signal ()

critical point

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

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.

3.6.5 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.

Listing 3.10: Reusable barrier hint

```
Semaphore turnstile = 0;
Semaphore turnstile2 = 1;
Semaphore mutex = 1;
```

Initally 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.

3.6.6 Reusable barrier solution

Listing 3.11: Reusable barrier solution

```
rendezvous

mutex.wait ()
    count++
    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--
    if (count == 0)
        turnstile.wait ()      // lock the first
        signal (turnstile2)    // unlock the second
mutex.signal ()

wait (turnstile2)              // second turnstile
signal (turnstile2)
```

This example is, unfortunately, typical of many synchronization problems: 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.

3.6.7 Barrier objects

It is natural to encapsulate a barrier in an object. The instance variables are `n`, `count`, `turnstile` and `turnstile2`.

Here is the code to initialize and use a `Barrier` object:

Listing 3.12: Barrier interface

```
Barrier barrier = 12      // initialize a new barrier
barrier.wait ()           // wait at a barrier
```

In this case the first 11 threads will block when they invoke `wait`. When the 12th thread arrives, all threads unblock. If the barrier object is implemented using the code in Section 3.5.4, the barrier object will be reusable; that is, after the first batch of 12 thread pass, the next batch has to wait until the 24th thread arrives.

3.6.8 Kingmaker pattern

As threads pass a barrier, it is sometimes useful to assign identifiers to the threads or to choose one of them to perform a special function. The counter inside the barrier provides an obvious way to solve this problem.

All we need is the ability to store a value in each thread. For example, each thread might have a variable named `threadNum` that records a different number for each thread. This kind of variable is called a **thread variable** or a **local variable** to distinguish it from the shared variables we have been using.

The following example assigns a different integer to each thread that passes. Here are the variable declarations:

Listing 3.13: Kingmaker initialization

```
Semaphore mutex = 1
int count = 0
local int threadNum
```

And here is the code:

Listing 3.14: Kingmaker code

```
mutex.wait ()
    count++
    threadNum = count
mutex.signal ()
```

Because `count` is a shared variable, any thread can increment it and all threads see the changed value. On the other hand, when a thread assigns a value to `threadNum`, only the local version of `threadNum` changes. The other threads are unaffected.

3.7 FIFO queue

If there is more than one thread waiting in queue when a semaphore is signalled, there is usually no way to tell which thread will be woken. Some implementations wake threads up in a particular order, like first-in-first-out, but the semantics of semaphores don't require any particular order. As an alternative, it might be useful to assign priorities to threads and wake them in order of priority.

But even if your environment doesn't provide first-in-first-out queueing, you can build it yourself.

Puzzle: use an array of semaphores to build a first-in-first-out queue that can hold up to `n` threads. Each time the semaphore `sem` is signalled, the thread at the head of the queue should proceed. You should not make any assumption about the implementation of your semaphores.

3.7.1 FIFO queue hint

Here are the variables I used.

Listing 3.15: FIFO queue hint

```
local int i
Semaphore array[n+1] = { 0, 1, 1, ..., 1 }
Semaphore sem = 0
```

The local variable `i` is used by each thread to keep track of where it is in queue. The array of semaphores controls the flow of threads through the queue. Assume that the elements of the array are indexed from 1 to `n+1`. Initially, the first semaphore is locked and all the rest are open. `sem` is the semaphore that is signalled to waken the thread at the head of the queue.

3.7.2 FIFO queue solution

Here is my solution:

Listing 3.16: FIFO queue solution

```
i = n
array[i+1].wait ()
while (i > 0) {
    array[i].wait ()
    array[i+1].signal ()
    i--
}
```

When a first thread arrives, it gets the `i+1` semaphore and then enters the loop. To advance in the queue, the thread has to get the next semaphore, `i`. If it succeeds, then it releases the previous semaphore, which allows a subsequent thread to proceed. If it fails, it waits while holding the previous semaphore, which blocks a subsequent thread.

The first thread to arrive will cascade down to `array[1]`, which is initially locked. When it blocks, it will hold `array[2]`. The next thread will block on `array[2]` while holding `array[3]`, and so on.

Eventually, some external thread will signal `sem` to allow a thread to exit the queue. To make that work, we can use a helper thread running this loop:

Listing 3.17: FIFO queue solution (helper)

```
while (1) {
    sem.wait ()
    array[1].signal ()
}
```

So, each time `sem` is signalled, the helper thread signals `array[1]`. That wakes up the first thread in queue, which will signal `array[2]` and then proceed. The second thread in queue will advance one position, which allows each of the following threads, in turn, to advance one position. At each step in the cascade, we know that there is only one thread waiting on each semaphore, and that each thread obtains the next semaphore before releasing the one it has. Therefore, no thread can overtake another and the threads exit the queue in FIFO order.

That is, unless the queue is full. In that case, more than one thread can be queued on `array[n+1]`, and we have no control over which one enters the queue next. So, to be precise, we can say that threads will leave the queue in the order they pass `array[n+1]`.

Naturally, we can encapsulate this solution in a object, called a `Fifo`, that has the following syntax:

Listing 3.18: FIFO queue solution (helper)

```
Fifo fifo(n)
fifo.wait ()
```

```
fifo.signal ()
```

The initializer creates a Fifo with `n` places in queue. `wait` and `signal` have the same semantics as for semaphores, except that threads proceed in the order they arrive, provided that there are no more than `n` in queue at a time.

Chapter 4

Classical synchronization problems

In this chapter we examine the classical problems that appear in nearly every operating systems class. 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. Producers 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:

Listing 4.1: Basic producer code

```
event = waitForEvent ()
addEventToBuffer (event)
```

Also, assume that consumers perform the following operations:

Listing 4.2: Basic consumer code

```
event = getEventFromBuffer ()
processEvent (event)
```

As specified above, access to the buffer has to be exclusive, but `waitForEvent` and `processEvent` 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:

Listing 4.3: Producer-consumer initialization

```
Semaphore mutex = 1
Semaphore length = 0
local Event event
```

Not surprisingly, `mutex` provides exclusive access to the buffer. When `length` 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 (each thread has its own variable with this name). Let's assume that capital-E `Event` is some type of object.

4.1.2 Producer-consumer solution

Here is the producer code from my solution.

Listing 4.4: Producer solution

```
event = waitForEvent ()
mutex.wait ()
    addEventToBuffer (event)
    length.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 `length` semaphore keeps track of the number of items in the buffer. Each time the producer adds an item, it signals `length`, incrementing it by one.

The consumer code is similar.

Listing 4.5: Consumer solution

```
length.wait ()
mutex.wait ()
    event = getEventFromBuffer ()
mutex.signal ()
processEvent (event)
```

Again, the buffer operation is protected by a mutex, but before the consumer gets to it, it has to decrement `length`. If `length` 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 `length`. 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:

Listing 4.6: Improved producer solution

```
event = waitForEvent ()
mutex.wait ()
    addEventToBuffer (event)
mutex.signal ()
length.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 `length` 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 `length` before any of them gets the mutex and removes an item from the buffer. At least for a little while, `length` would be inaccurate.

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

Listing 4.7: Broken consumer solution

```
mutex.wait ()
    length.wait ()
    event = getEventFromBuffer ()
mutex.signal ()
processEvent (event)
```

This is a bad idea.

Puzzle: why?

4.1.3 Deadlock #4

If the consumer is running this code

Listing 4.8: Broken consumer solution

```
mutex.wait ()
    length.wait ()
    event = getEventFromBuffer ()
mutex.signal ()

processEvent (event)
```

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

Listing 4.9: Broken finite buffer solution

```
if (length >= bufferSize)
    block ()
```

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 constraint.

4.1.5 Finite buffer producer-consumer hint

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

Listing 4.10: Finite-buffer producer-consumer initialization

```
Semaphore mutex = 1
Semaphore length = 0
Semaphore spaces = bufferSize
```

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.

Listing 4.11: Finite buffer consumer solution

```
length.wait ()
mutex.wait ()
    event = getEventFromBuffer ()
mutex.signal ()
spaces.signal ()

processEvent (event)
```

The producer code is symmetric, in a way:

Listing 4.12: Finite buffer producer solution

```
event = waitForEvent ()

spaces.wait()
mutex.wait ()
    addEventToBuffer (event)
mutex.signal ()
length.signal ()
```

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

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 basic 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.

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.

Listing 4.13: Readers-writers initialization

```
int readers = 0;
Semaphore mutex = 1;
Semaphore roomEmpty = 1;
```

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.

4.2.2 Readers-writers solution

The code for writers is simple. If the critical section is empty, a writer may enter, but entering has the effect of excluding all other threads:

Listing 4.14: Writers solution

```
roomEmpty.wait ()
    critical section for writers
roomEmpty.signal ()
```

When the writer exits, can it be sure that the room is now empty? Yes, because it knows that no other thread can have entered while it was there.

The code for readers is similar to the barrier code we saw in the previous section. We keep track of the number of readers in the room so that we can give a special assignment to the first to arrive and the last to leave.

The first reader that arrives has to wait for `roomEmpty`. If the room is empty, then the reader proceeds and, at the same time, bars writers. Subsequent readers can still enter because none of them will try to wait on `roomEmpty`.

If a reader arrives while there is a writer in the room, it waits on `roomEmpty`. Since it holds the mutex, any subsequent readers queue on `mutex`.

Listing 4.15: Readers solution

```
mutex.wait ()
    readers++
    if (readers == 1)
        roomEmpty.wait ()   // first in locks
mutex.signal ()

    critical section for readers

mutex.wait ()
    readers--
    if (readers == 0)
        roomEmpty.signal () // last out unlocks
mutex.signal ()
```

The code after the critical section is similar. The last reader to leave the room turns out the lights—that is, it signals `roomEmpty`, possibly allowing a waiting writer to enter.

Again, to demonstrate that this code is correct, it is useful to assert and demonstrate a number of claims about how the program must behave. Can you convince yourself that the following are true?

- Only one reader can queue waiting for `roomEmpty`, but several writers might be queued.
- When a reader signals `roomEmpty` the room must be empty.

Patterns similar to this reader code are common: the first thread into a section locks a semaphore (or queues) and the last one out unlocks it. In fact, it is so common we should give it a name and wrap it up in an object.

The name of the pattern is "Film Louie," which stands for "first in locks mutex, last out unlocks it, eh?" For convenience, we'll call the pattern a Louie and exapsulate it in an object, `Louie`, with the following instance variable:

Listing 4.16: Louie definition

```
int counter
Semaphore mutex
```

The methods that operate on `Louie`s are `lock` and `unlock`.

Listing 4.17: Louie lock definition

```
function lock (semaphore)
    wait (mutex)
        counter++
        if (counter = 1)
            wait (semaphore)
    signal (mutex)
```

`lock` takes one parameter, a semaphore that it will check and possibly hold. It performs the following operations:

- If the semaphore is locked, then the calling thread and all subsequent callers will block.
- When the semaphore is signalled, all waiting threads and all subsequent callers can proceed.
- While the filo is locked, the semaphore is locked.

Inside the definition of `lock`, the references to `mutex` and `counter` refer to the instance variables of the current object, the `Louie` the method was invoked on.

Here is the code for `unlock`:

Listing 4.18: Louie unlock definition

```
function filoUnlock (semaphore)
    wait (mutex)
        counter--
        if (counter = 0)
            signal (semaphore)
    signal (mutex)
```

`unlock` has the following behavior:

- This function does nothing until every thread that called `lock` also calls `unlock`.
- When the last thread calls `unlock`, it unlocks the semaphore.

Using these functions, we can rewrite the reader code more simply:

Listing 4.19: Concise reader solution

```
readLouie.lock (roomEmpty)
  critical section for readers
readLouie.unlock (roomEmpty)
```

`readLouie` is a shared `Louie` object whose counter is initially zero.

4.2.3 Starvation

In the previous solution, is there any danger of deadlock? In order for a deadlock to occur, it must be possible for a thread to wait on a semaphore while holding another, and thereby prevent itself from being signalled.

In this example, deadlock is not possible, but there is a related problem that is almost as bad: it is possible for a writer to starve.

If a writer arrives while there are readers in the critical section, it might wait in queue forever while readers come and go. As long as a new reader arrives before the last of the current readers exits, there will always be at least one reader in the room.

This situation is not a deadlock, because some threads are making progress, but it is not exactly desireable. A program like this might work as long as the load on the system is low, because then there are plenty of opportunities for the writers. But as the load increases the behavior of the system would deteriorate quickly (at least from the point of view of writers).

Puzzle: Extend this solution so that when a writer arrives, the existing readers can finish, but no additional readers may enter.

4.2.4 No-starve readers-writers hint

Here's a hint. You can add a turnstile for the readers and allow writers to lock it. The writers have to pass through the same turnstile, but they should check the `roomEmpty` semaphore while they are inside the turnstile. If a writer gets stuck in the turnstile it has the effect of forcing the readers to queue at the turnstile. Then when the last reader leaves the critical section, we are guaranteed that at least one writer enters next (before any of the queued readers can proceed).

Listing 4.20: No-starve readers-writers initialization

```
Louie readers = 0
Semaphore roomEmpty = 1
Semaphore turnstile = 1
```

`readers` keeps track of how many readers are in the room; the `Louie` locks `roomEmpty` when the first reader enters and unlocks it when the last reader leaves.

`turnstile` is a turnstile for readers and a mutex for writers.

4.2.5 No-starve readers-writers solution

Here is the writer code:

Listing 4.21: No-starve writer solution

```
1 turnstile.wait ()
2
3     roomEmpty.wait ()
4
5     critical section for writers
6
7 turnstile.signal ()
8
9 roomEmpty.signal ()
```

If a writer arrives while there are readers in the room, it will block at Line 3, which means that the turnstile will be locked. This will bar readers from entering while a writer is queued. Here is the reader code:

Listing 4.22: No-starve reader solution

```
turnstile.wait ()
turnstile.signal ()

filo.lock (roomEmpty)
    critical section for readers
filo.unlock (roomEmpty)
```

When the last reader leaves, it signals `roomEmpty`, unblocking the waiting writer. The writer immediately enters its critical section, since none of the waiting readers can pass the turnstile.

When the writer exits it signals `turnstile`, which unblocks a waiting thread, which could be a reader or a writer. Thus, this solution guarantees that at least one writer gets to proceed, but it is still possible for a reader to enter while there are writers queued.

Depending on the application, it might be a good idea to give more priority to writers. For example, if writers are making time-critical updates to a data structure, it is best to minimize the number of readers that see the old data before the writer has a chance to proceed.

In general, though, it is up to the scheduler, not the programmer, to choose which waiting thread to unblock. Some schedulers use a first-in-first-out queue, which means that threads are unblocked in the same order they queued. Other schedulers choose at random, or according to a priority scheme based on the properties of the waiting threads.

If your programming environment makes it possible to give some threads priority over others, then that is a simple way to address this issue. If not, you will have to find another way.

Puzzle: Write a solution to the readers-writers problem that gives priority to writers. That is, once a writer arrives, no readers should be allowed to enter until all writers have left the system.

4.2.6 Writer-priority readers-writers hint

As usual, the hint is in the form of variables used in the solution.

Listing 4.23: Writer-priority readers-writers initialization

```
Louie readLouie, writeLouie
Semaphore mutex
Semaphore noReaders = 1
Semaphore noWriters = 1
```

4.2.7 Writer-priority readers-writers solution

Here is the reader code:

Listing 4.24: Writer-priority reader solution

```
noReaders.wait()
    readLouie.lock (noWriters)
noReaders.signal ()

    critical section for readers
readLouie.unlock (noWriters)
```

If a reader is in the critical section, it holds `noWriters`, but it doesn't hold `noReaders`. Thus if a writer arrives it can lock `noReaders`, which will cause subsequent readers to queue.

When the last reader exits, it signals `noWriters`, allowing any queued writers to proceed.

The writer code:

Listing 4.25: Writer-priority writer solution

```
writeLouie.lock (noReaders)
    noWriters.wait ()
        critical section for writers
    noWriters.signal ()
writeLouie.unlock (noReaders)
```

When a writer is in the critical section it holds both `noReaders` and `noWriters`. This has the (relatively obvious) effect of insuring that there are no readers and no other writers in the critical section. In addition, `writeLouie` has the (less obvious) effect of allowing multiple writers to queue on `noWriters`, but keeping `noReaders` locked while they are there. Thus, many writers can pass through the critical section without without ever signalling `noReaders`. Only when the last writer exits can the readers enter.

Of course, a drawback of this solution is that now it is possible for *readers* to starve (or at least face long delays). For some applications it might be better to get obsolete data with predictable turnaround times.

Also, this solution does not quite satisfy all the requirements. While there is a writer in the critical section, it is possible for multiple readers to queue on `noReaders`. If a writer arrives, it will queue on `noReaders` along with the waiting readers. When the first writer leaves the critical section, the scheduler might allow the waiting readers to go ahead of the writer.

Again, depending on the application, that possibility might be tolerable. If not, we can improve the situation by adding a mutex to the reader code:

Listing 4.26: Writer-priority improved reader solution

```
mutex.wait ()
   noReaders.wait()
      readLouie.lock (noWriters)
   noReaders.signal ()
mutex.signal ()

   critical section for readers
readLouie.unlock (noWriters)
```

Now there can be at most one reader queued on `noReaders`. If a writer arrives, it will have to wait for any readers in the critical section, and it might have to wait for the queued reader, but that's all.

You might want to keep this trick in mind. To limit the number of threads in a certain queue, surround it with a mutex or a multiplex.

4.3 No-starve mutex

In the previous section, we addressed what I'll call **categorical starvation**, in which one category of threads (readers) allows another category (writers) to starve. At a more basic level, we have to address the issue of **thread starvation**, which is the possibility that one thread might wait indefinitely while others proceed.

For most concurrent applications, starvation is unacceptable, so we enforce the requirement of **bounded waiting**, which means that the time a thread waits on a semaphore (or anywhere else, for that matter) has to be provably finite.

In part, starvation is the responsibility of the scheduler. Whenever multiple threads are ready to run, the scheduler decides which one or, on a parallel processor, which set of threads gets to run. If a thread is never scheduled, then it will starve, no matter what we do with semaphores.

So in order to say anything about starvation, we have to start with some assumptions about the scheduler. If we are willing to make a strong assumption, we can assume that the scheduler uses one of the many algorithms that can be proven to enforce bounded waiting. If we don't know what algorithm the scheduler uses, then we can get by with a weaker assumption:

> Property 1: if there is only one thread that is ready to run, the scheduler has to let it run.

If we can assume Property 1, then we can build a system that is provably free of starvation. For example, if there are a finite number of threads, then any program that contains a barrier cannot starve, since eventually all threads but one will be waiting at the barrier, at which point the last thread has to run.

In general, though, it is non-trivial to write programs that are free from starvation unless with make the stronger assumption:

> Property 2: if a thread is ready to run, then the time it waits until it runs is bounded.

In our discussion so far, we have been assuming Property 2 implicitly, and we will continue to. On the other hand, you should know that many existing systems use schedulers that do not guarantee this property strictly.

Even with Property 2, starvation rears its ugly head again when we introduce semaphores. In the definition of a semaphore, we said that when one thread executes `signal`, one of the waiting threads gets woken up. But we never said which one. Again, in order to say anything about starvation, we have to make assumptions about the behavior of semaphores.

The weakest assumption that makes it possible to avoid starvation is:

> Property 3: if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.

This requirement may seem obvious, but it is not trivial. It has the effect of barring one form of problematic behavior, which is a thread that signals a semaphore while other threads are waiting, and then keeps running, waits on the same semaphore, and gets its own signal! If that were possible, that there would be nothing we could do to prevent starvation.

With Property 3, it becomes possible to avoid starvation, but even for sometime as simple as a mutex, it is not easy. For example, imagine three threads running the following code:

Listing 4.27: Mutex loop

```
while (1) {
    mutex.wait ()
    critical section
    mutex.signal ()
}
```

The `while` statement is an infinite loop; in other words, as soon as a thread leaves the critical section, it loops to the top and tries to get the mutex again.

Imagine that Thread A gets the mutex and Thread B and C wait. When A leaves, B enters, but before B leaves, A loops around and joins C in the queue. When B leaves, there is no guarantee that C goes next. In fact, if A goes next, and B joins the queue, then we are back to the starting position, and we can repeat the cycle forever. C starves.

The existence of this pattern proves that the mutex is vulnerable to starvation. One solution to this problem is to change the implementation of the semaphore so that it guarantees a stronger property:

> Property 4: if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.

For example, if the semaphore maintains a first-in-first-out queue, then Property 4 holds because when a thread joins the queue, the number of threads ahead of it is finite, and no threads that arrive later can go ahead of it.

A semaphore that has Property 4 is sometimes called a **strong semaphore**; one that has only Property 3 is called a **weak semaphore**. We have shown that with weak semaphors, the simple mutex solution is vulnerable to starvation. In fact, Dijkstra conjectured that it is not possible to solve the mutex problem without starvation using only weak semaphores.

In 1979, J.M. Morris refuted the conjecture by solving the problem, assuming that the number of threads is finite [4]. If you are interested in this problem, the next section presents his solution. If this is not your idea of fun, you can just assume that semaphores have Property 4 and go on to Section 4.4.

Puzzle: write a solution to the mutual exclusion problem using weak semaphores. Your solution should provide the following guarantee: once a thread arrives and attempts to enter the mutex, there is a bound on the number of threads that can proceed ahead of it.

4.3.1 No-starve mutex hint

Morris's solution is similar to the reusable barrier in Section 3.6. It uses two turnstiles to create two waiting rooms before the critical section. The mechanism works in two phases. During the first phase, the first turnstile is open and the second is closed, so threads accumulate in the second room. During the second phase, the first turnstile is closed, so no new threads can enter, and the second is open, so the existing threads can get to the critical section.

Although there may be an arbitrary number of threads in the waiting room, each one is guaranteed to enter the critical section before any future arrivals.

Here are the variables I used in the solution (I changed the names Morris uses to make the structure clearer).

Listing 4.28: No-starve mutex hint

```
int room1, room2
Semaphore mutex
Semaphore t1 = 1
Semaphore t2 = 0
```

`room1` and `room2` keep track of how many threads are in the waiting rooms. `mutex` helps protect the counters. `t1` and `t2` are the turnstiles.

4.3.2 No-starve mutex solution

Here is Morris's solution.

Listing 4.29: Morris's algorithm

```
1  mutex.wait ()
2      room1++
3  mutex.signal ()
4
5  t1.wait ()
6      room2++
7      mutex.wait ()
8      room1--
9
10     if (room1 == 0)
11         mutex.signal ()
12         t2.signal ()
13     else
14         mutex.signal ()
15         t1.signal ()
16
17 t2.wait ()
18     room2--
19
20     // critical section
21
22     if (room2 == 0)
23         t1.signal ()
24     else
25         t2.signal ()
```

Before entering the critical section, a thread has to pass two turnstiles. These turnstiles divide the code into three "rooms". Room 1 is Lines 1–5. Room 2 is lines 6–17. Room 3 is the rest. Speaking loosely, the counters `room1` and `room2` keep track of the number of threads in each room.

The counter `room1` is protected by `mutex` in the usual way, but guard duty for `room2` is split between `t1` and `t2`. Similarly, responsibility for exclusive access to the critical section involves both `t1` and `t2`. In order to enter the critical section, a thread has to hold one or the other, but not both. Then, before exiting, it gives up whichever one it has.

To understand how this solution works, start by following a single thread all the way through. When it gets to Line 10, it holds `mutex` and `t1`. Once it checks `room1`, which is 0, it can release `mutex` and then open the second turnstile, `t2`. As a result, it doesn't wait at Line 17 and it can safely decrement `room2` and enter the critical section, because any following threads have to be queued on `t1`. Leaving the critical section, it finds `room2 = 0` and releases `t1`, which brings us back to the starting state.

Of course, the situation is more interesting if there is more than one thread. In that case, it is possible that when the lead thread gets to Line 10, other threads have entered the waiting room and queued on `t1`. Since `room1 > 0`, the lead thread leaves `t2` locked and instead signals `t1` to allow a following thread to enter Room 2. Since `t2` is still locked, neither thread can enter Room 3.

Eventually (because there are a finite number of threads), a thread will get to Line 10 before another threads enters Room 1, in which case it will open `t2`, allowing any threads there to move into Room 3. The thread that opens `t2` continues to hold `t1`, so if any of the lead threads loop around, they will block at Line 5.

As each thread leaves Room 3, it signals `t2`, allowing another thread to leave Room 2. When the last thread leaves Room 2, it leaves `t2` locked and opens `t1`, which brings us back to the starting state.

To see how this solution avoids starvation, it helps to think of its operation in two phases. In the first phase, threads check into Room 1, increment `room1`, and then cascade into Room 2 one at a time. The only way to keep a thread from proceeding is to maintain a procession of threads through Room 1. Because there are a finite number of threads, the procession has to end eventually, at which point the first turnstile locks and the second opens.

In the second phase, threads cascade into Room 3. Because there are a finite number of threads in Room 2, and no new threads can enter, eventually the last thread leaves, at which point the second turnstile locks and the first opens.

At the end of the first phase, we know that there are no threads waiting at `t1`, because `room1 = 0`. And at the end of the second phase, we know that there are no threads waiting at `t2` because `room2 = 0`.

With a finite number of threads, starvation is only possible if one thread can loop around and overtake another. But the two-turnstile mechanism makes that impossible, so starvation is impossible.

The moral of this story is that with weak semaphores, it is very difficult to prevent starvation, even for the simplest synchonization problems. In the rest of this book, when we discuss starvation, we will assume strong semaphores.

4.4 Dining philosophers

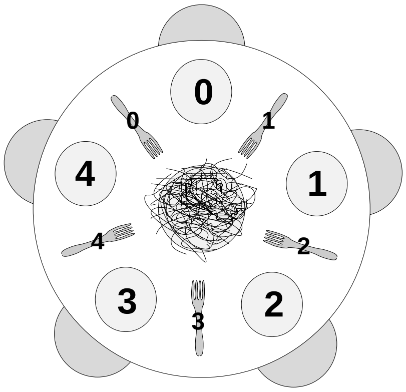
The Dining Philosophers Problem was proposed by Dijkstra in 1965, when dinosaurs ruled the earth [2]. It appears in a number of variations, but the standard features are a table with five plates, five forks (or chopsticks) and a big bowl of spaghetti. Five philosophers, who represent interacting threads, come to the table and execute the following loop:

Listing 4.30: Basic philosopher loop

```
while (1) {
   think ()
   get_forks ()
   eat ()
   put_forks ()
}
```

The forks represent resources that the threads have to hold exclusively in order to make progress. The thing that makes the problem interesting, unrealistic, and unsanitary, is that the philosophers need *two* forks to eat, so a hungry philosopher might have to wait for a neighbor to put down a fork.

Assume that the philosophers have a local variable `i` that identifies each philosopher with a value in (0..4). Similarly, the forks are numbered from 0 to 4, so that Philosopher i has fork i on the right and fork $(i+1) mod 5$ on the left. Here is a diagram of the situation:



Assuming that the philosophers know how to `think` and `eat`, our job is to write a version of `get_forks` and `put_forks` that satisfies the following constraints:

- Only one philosopher can hold a fork at a time.
- It must be impossible for a deadlock to occur.
- It must be impossible for a philosopher to starve waiting for a fork.
- It must be possible for more than one philosopher to eat at the same time.

The last requirement is one way of saying that the solution should be efficient; that is, it should allow the maximum amount of concurrency.

We make no assumption about how long `eat` and `think` take, except that `eat` has to terminate eventually. Otherwise, the third constraint is impossible—if a philosopher keeps one of the forks forever, nothing can prevent the neighbors from starving.

To make it easy for philosophers to refer to their forks, we can use the functions `left` and `right`:

Listing 4.31: Which fork?

```
int left (i) { return i }
int right (i) { return (i + 1) mod 5 }
```

The `mod` operator wraps around when it gets to 5, so that `4 + 1 = 0`.

Since we have to enforce exclusive access to the forks, it is natural to use an array of Semaphores, one for each fork. Initially, all the forks are available.

Listing 4.32: Variables for dining philosophers

```
Semaphore fork[5] = {1, 1, 1, 1, 1}
```

Here is an initial attempt at `get_fork` and `put_fork`:

Listing 4.33: Dining philosophers non-solution

```
get_fork (i) {
    fork[right(i)].wait()
    fork[left(i)].wait()
}

put_fork (i) {
    fork[right(i)].signal()
    fork[left(i)].signal()
}
```

It's clear that this solution satisfies the first constraint, but we can be pretty sure it doesn't satisfy the other two, because if it did, this wouldn't be an interesting problem and you would be reading Chapter 5.

Puzzle: what's wrong?

4.4.1 Deadlock #5

The problem is that the table is round. As a result, each philosopher can pick up a fork and then wait forever for the other fork. Deadlock!

Puzzle: write a solution to this problem that prevents deadlock.

Hint: one way to avoid deadlock is to think about the conditions that make deadlock possible and then change one of those conditions. In this case, the deadlock is fairly fragile—a very small change breaks it.

4.4.2 Dining philosophers hint #1

If only four philosophers are allowed at the time at a time, deadlock is impossible.

First, convince yourself that this claim is true, then write code that limits the number of philosophers at the table.

4.4.3 Dining philosophers solution #1

If there are only four philosophers at the table, then in the worst case each one picks up a fork. Even then, there is a fork left on the table, and that fork has two neighbors, each of which is holding another fork. Therefore, either of these neighbors can pick up the remaining fork and eat.

We can control the number of philosophers at the table with a Multiplex named `footman` that is initialized to 4. Then the solution looks like this:

Listing 4.34: Dining philosophers solution #1

```
get_fork (i) {
    footman.wait ()
    fork[right(i)].wait()
    fork[left(i)].wait()
}

put_fork (i) {
    fork[right(i)].signal()
    fork[left(i)].signal()
    footman.signal ()
}
```

Can we prove that this solution prevents starvation? Imagine that you are sitting at the table and both of your neighbors are eating. You are blocked waiting for your right fork. Eventually your right neighbor will put it down, because `eat` can't run forever. Since you are the only thread waiting for that fork, you will necessarily get it next. By a similar argument, you cannot starve waiting for your left fork.

Therefore, the time a philosopher can spend at the table is bounded. That implies that the wait time to get into the room is also bounded, as long as `footman` has Property 4 (see Section 4.3).

This solution shows that by controlling the number of philosophers, we can avoid deadlock. Another way to avoid deadlock is to change the order in which the philosophers pick up forks. In the original non-solution, the philosophers are "righties"; that is, they pick up the right fork first. But what happens if Philosopher 0 is a leftie?

Puzzle: prove that if at least one Philosopher is a leftie, then deadlock is not possible.

Hint: deadlock can only occur when all 5 philosophers are holding one fork and waiting, forever, for the other. Otherwise, one of them could get both forks, eat, and leave.

The proof works by contradiction. First, assume that deadlock is possible. Then choose one of the supposedly deadlocked philosophers. If she's a leftie, you can prove that the philosophers are all lefties, which is a contradiction. Similarly, if she's a rightie, can prove that they are all righties. Either way you get a contradiction; therefore, deadlock is not possible.

4.4.4 Dining philosopher's solution #2

In the asymmetric solution to the Dining philosophers problem, there has to be at least one leftie and at least one rightie at the table. In that case, deadlock is impossible. The previous hint outlines the proof. Here are the details.

Again, if deadlock is possible, it occurs when all 5 philosophers are holding one fork and waiting for the other. If we assume that Philosopher j is a leftie, then she must be holding her left fork and waiting for her right. Therefore her neighbor to the right, Philosopher k, must be holding his left fork and waiting for his right neighbor; in other words, Philosopher k must be a leftie. Repeating the same argument, we can prove that the philosophers are all lefties, which contradicts the original claim that there is at least one rightie. Therefore deadlock is not possible.

The same argument we used for the previous solution also proves that starvation is impossible for this solution.

4.4.5 Tanenbaum's solution

There is nothing wrong with the previous solutions, but just for completeness, let's look at some alternatives. One of the best known is the one that appears in Tanenbaum's popular operating systems textbooks [9]. For each philosopher there is a state variable that indicates whether the philosopher is thinking, eating, or waiting to eat ("hungry") and a semaphore that indicates whether the philosopher can start eating. Here are the variables:

Listing 4.35: Variables for Tanenbaum's solution

```
int state[5]
Semaphore sem[5]
Semaphore mutex = 1
```

The state array is initialized to "thinking," and the semaphore array is initialized to 0. Here is the code:

Listing 4.36: Tanenbaum's solution

```
get_fork (i) {
    mutex.wait ()
    state[i] = hungry
    test (i)
    mutex.signal ()
    sem[i].wait ()
}

put_fork (i) {
    mutex.wait ()
    state[i] = thinking
    test (right (i))
    test (left (i))
    mutex.signal ()
}

test (i) {
    if (state[i] == hungry &&
    state (left (i)) != eating &&
    state (right (i)) != eating)
        state[i] = eating
        sem[i].signal ()
}
```

The `test` function checks whether the ith philosopher can start eating, which he can if he is hungry and neither of his neighbors are eating. If so, the `test` signals semaphore i.

There are two ways a philosopher gets to eat. In the first case, the philosopher executes `get_forks`, finds the forks available, and proceeds immediately.

In the second case, one of the neighbors is eating and the philosopher blocks on its own semaphore. Eventually, one of the neighbors will finish, at which point it executes `test` on both of its neighbors. It is possible that both tests will succeed, in which case the neighbors can run concurrently. The order of the two tests doesn't matter.

In order to access the state array, or invoke `test`, a thread has to hold `mutex`. Thus, the operation of checking and updating the array is atomic. Since a philosopher can only proceed when we know both forks are available, exclusive access to the forks is guaranteed.

No deadlock is possible, because the only semaphore that is accessed by more than one philosopher is `mutex`, and no thread executes `wait` while holding `mutex`.

But again, starvation is tricky.

Puzzle: Either convince yourself that Tanenbaum's solution prevents starvation or find a repeating pattern that allows a thread to starve while other threads make progress.

4.4.6 Starving Tanenbaums

Unfortunately, this solution is not starvation-free. Gingras demonstrated that there are repeating patterns that allow a thread to wait forever while other threads come and go [3].

Imagine that we are trying to starve Philosopher 0. Initially, 2 and 4 are at the table and 1 and 3 are hungry. If 4 gets up first, then only 0 can proceed, so we'll have 2 get up and 1 sit down. Since Philosopher 1 holds fork 1, it is now "safe" to let 4 get up and 3 sit down.

Now we are in the mirror image of the starting position. Again, if 1 got up first, 0 could eat, so instead 3 gets up and 4 sits down. Finally, 1 gets up and 2 sits down, which brings us back where we started. We could repeat the cycle indefinitely and Philosopher 1 would starve.

So, Tanenbaum's solution doesn't satisfy all the requirements.

4.5 Cigarette smokers' problem

The cigarette smokers' problem problem was originally presented by Suhas Patil [6], who claimed that it cannot be solved with semaphores. That claim comes with some qualifications, but in any case the problem is interesting and challenging.

Four threads are involved: an agent and three smokers. The smokers loop forever, first waiting for ingredients, then making and smoking cigarettes. The ingredients are tobacco, paper, and matches.

We assume that the agent has an infinite supply of all three ingredients, and each smoker has an infinite supply of one of the ingredients; that is, one smoker has matches, another has paper, and the third has tobacco.

The agent repeatedly chooses two different ingredients at random and makes them available to the smokers. Depending on which ingredients are chosen, the smoker with the complementary ingredient should pick up both resources and proceed.

For example, if the agent puts out tobacco and paper, the smoker with the matches should pick up both ingredients, make a cigarette, and then signal the agent.

Based on this premise, there are three versions of this problem that often appear in textbooks:

The impossible version: Patil's version imposes several artificial restrictions on the solution. First, the code for the agent and a set of accompanying semaphores is given as an unmodifiable part of the problem. Second, the solution is not allowed to use conditional statements or an array of semaphores. With these constraints, the problem cannot be solved. In a followup article, Parnas points out that these restrictions are uncomfortably artificial [5]. With constraints like these, a lot of problems become unsolvable.

The interesting version: This version imposes no additional constraints other than the usual goals: all threads should make progress (no starvation) and no threads should busy-wait (check for a condition over and over).

The trivial version: In some textbooks, the problem specifies that the agent should signal the smoker that should go next, according to the ingredients that are available. This version of the problem is uninteresting because it makes the whole premise, the ingredients and the cigarettes, irrelevant. Also, as a practical matter, it is probably not a good idea to require the agent to know about the other threads and what they are waiting for. Finally, this version of the problem is just too easy.

Naturally, we will focus on the interesting version. To complete the statement of the problem, we need to specify the agent code. The agent uses the following semaphores:

Listing 4.37: Agent semaphores

```
Semaphore agentSem = 1
Semaphore tobacco = 0
Semaphore paper = 0
Semaphore match = 0
```

After putting out two ingredients, the agent waits for `agentSem` before continuing. The other three semaphores indicate which ingredients are available.

The agent is actually made up of three concurrent threads, Agent A, Agent B and Agent C. Each provides different ingredients.

Listing 4.38: Agent A code

```
agentSem.wait ()
tobacco.signal ()
paper.signal ()
```

Listing 4.39: Agent B code

```
agentSem.wait ()
paper.signal ()
match.signal ()
```

Listing 4.40: Agent C code

```
agentSem.wait ()
tobacco.signal ()
match.signal ()
```

Initally the three agent threads compete for `agentSem`. Whichever gets it determines the first pair of ingredients. Subsequently, each time `agentSem` is signalled, a “random” agent thread proceeds. Of course, the behavior of the system is not really random; it depends on the decisions of the local scheduler (which thread runs when) and the queueing policy of the semaphore.

This problem is hard because the natural solution does not work. It is tempting to write something like:

Listing 4.41: Smoker with matches

```
tobacco.wait ()
paper.wait ()
agentSem.signal ()
```

Listing 4.42: Smoker with tobacco

```
paper.wait ()
match.wait ()
agentSem.signal ()
```

Listing 4.43: Smoker with paper

```
tobacco.wait ()
match.wait ()
agentSem.signal ()
```

What's wrong with this solution?

4.5.1 Deadlock #6

The problem with the previous solution is the possibility of deadlock. Imagine that the agent puts out tobacco and paper. Since the smoker with matches is waiting on `tobacco`, it might be unblocked. But the smoker with tobacco is waiting on `paper`, so it is possible (even likely) that it will also be unblocked. Then the first thread will block on `paper` and the second will block on `match`. Deadlock!

4.5.2 Smoker problem hint

The solution proposed by Parnas uses three helper threads called "pushers" that respond to the signals from the agent, keep track of the available ingredients, and signal the appropriate smoker.

The additional variables and semaphores are

Listing 4.44: Smoker problem hint

```
boolean isTobacco, isPaper, isMatch
Semaphore tobaccoSem = 0
Semaphore paperSem = 0
Semaphore matchSem = 0
```

The boolean variables indicate whether or not an ingredient is on the table. The pushers use `tobaccoSem` to signal the smoker with tobacco, and the other semaphores likewise.

4.5.3 Smoker problem solution

Here is the code for one of the pushers:

Listing 4.45: Pusher A

```
tobacco.wait ()
mutex.wait ()
    if (isPaper)
        isPaper = false
        matchSem.signal ()
    else if (isMatch)
        isMatch = false
        paperSem.signal ()
    else
        isTobacco = true
mutex.signal ()
```

This pusher wakes up any time there is tobacco on the table. If it finds `isPaper` true, it knows that Pusher B has already run, so it can signal the smoker with matches. Similarly, if it finds a match on the table, it can signal the smoker with paper.

But if Pusher A runs first, then it will find both `isPaper` and `isMatch` false. It cannot signal any of the smokers, so it sets `isTobacco`.

The other pushers are similar. Since the pushers do all the real work, the smoker code is trivial:

Listing 4.46: Smoker with tobacco

```
tobaccoSem.wait ()
makeCigarette ()
agentSem.signal ()
smoke ()
```

Parnas presents a similar solution that assembles the boolean variables, bitwise, into an integer, and then uses the integer as an index into an array of semaphores. That way he can avoid using conditionals (one of the artificial constraints). The resulting code is a bit more concise, but its function is not as obvious.

4.5.4 Generalized Smokers' Problem

Parnas suggested that the smokers' problem becomes more difficult if we modify the agent, eliminating the requirement that the agent wait after putting out ingredients. In this case, there might be multiple instances of an ingredient on the table.

Puzzle: modify the previous solution to deal with this variation.

Chapter 5

Not-so-classical synchronization problems

5.1 Building H_2O

This problem has been a staple of the Operating Systems class at U.C. Berkeley for at least a decade. It seems to be based on an exercise in Andrews's *Concurrent Programming* [1].

There are two kinds of threads, oxygen and hydrogen. In order to assemble these threads into water molecules, we have to create a barrier that makes each thread wait until a complete molecule is ready to proceed.

As each thread passes the barrier, it should invoke a method called `bond`. You must guarantee that all the threads from one molecule invoke `bond` before any of the threads from the next molecule do.

In other words:

- If an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads.
- If a hydrogen thread arrives at the barrier when no other threads are present, it has to wait for an oxygen thread and another hydrogen thread.
- If we examine the sequence of threads that invoke `bond` and divide them into groups of three, each group will contain one oxygen and two hydrogen threads.

We don't have to worry about matching the threads up explicitly; that is, the threads do not necessarily know which other threads they are paired up with. The key is just that threads pass the barrier in complete sets.

Puzzle: Write synchronization code for oxygen and hydrogen molecules.

5.1.1 H_2O hint

Here are the variables I used in my solution:

Listing 5.1: Water building hint

```
Semaphore hydroSem = 0
Semaphore hydroSem2 = 0
Semaphore oxySem = 0
Semaphore mutex = 1
int count = 0
```

Hydrogen threads have to wait twice, first on `hydroSem`, later on `hydroSem2`; oxygen threads wait on `oxySem`. `count` keeps track of how many hydrogen threads are waiting for oxygen.

5.1.2 H_2O solution

Initially `hydroSem` and `oxySem` are locked. When an oxygen thread arrives it signals `hydroSem` twice, allowing two hydrogens to proceed. Then the oxygen thread waits for the hydrogen threads to arrive.

Listing 5.2: Oxygen code

```
signal (hydroSem)
signal (hydroSem)

wait (oxyMutex)
    wait (oxySem)
    bond ()
signal (oxyMutex)
```

In order for a hydrogen thread to pass Line 1, there must be an oxygen thread waiting. The mutex-protected counter keeps track of how many hydrogen threads have arrived. When `count` gets to 2, we signal `oxySem` once and `hydroSem2` twice.

Listing 5.3: Hydrogen code

```
 1 wait (hydroSem)
 2 mutex.wait ()
 3     count++
 4     if (count == 2)
 5         signal (oxySem)
 6         signal (hydroSem2)
 7         signal (hydroSem2)
 8         count = 0
 9 mutex.signal ()
10
11 wait (hydroSem2)
12 bond ()
```

To demonstrate the correctness of this solution, consider the following medium-level assertions:

- For every molecule, we need to signal `oxySem` once and both hydrogen semaphores twice.
- Every oxygen thread signals `hydroSem` twice.
- In each pair of hydrogen threads, one of them signals `oxySem` once and `hydroSem2` twice.

There is one aspect of this solution that is disconcerting but that may not be problematic. Hydrogen threads do not necessarily invoke `bond` in the same order they pass through `hydroSem`. In particular, an oxygen thread might signal two hydrogen threads and then end up bonding with two different hydrogen threads. That outcome does not violate the problem constraints as stated, but at the same time it doesn't seem quite right.

If you are bothered by this problem, rewrite the solution to fix it. You might want to do the next problem first and then come back.

5.2 River crossing problem

This problem is from a problem set written by Anthony Joseph at U.C. Berkeley, but I don't know if he is the original author. It is similar to the H_2O problem in the sense that it is a peculiar sort of barrier that only allows threads to pass in certain combinations.

Somewhere near Redmond, Washington there is a rowboat that is used by both Linux hackers and Microsoft employees (serfs) to cross a river. The ferry can hold exactly four people; it won't leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe.

As each thread boards the boat it should invoke a function called `board`. You must guarantee that all four threads from each boatload invoke `board` before any of the threads from the next boatload do.

After all four threads have invoked `board`, exactly one of them should call a function named `rowBoat`, indicating that that thread will take the oars. It doesn't matter which thread calls the function, as long as one does.

Don't worry about the direction of travel. Assume we are only interested in traffic going in one of the directions.

5.2.1 River crossing hint

Here are the variables I used in my solution:

Listing 5.4: River crossing hint

```
Semaphore mutex = 1
Barrier barrier = 4
int hackers = 0
int serfs = 0
Semaphore hackerSem = 0
Semaphore serfSem = 0
local boolean iAmTheCaptain = false
```

`hackers` and `serfs` count the number of hackers and serfs waiting to board. Since they are both protected by `mutex`, we can check the condition of both variables without worrying about an untimely update.

`hackerSem` and `serfSem` allow us to control the number of hackers and serfs that pass. The barrier makes sure that all four threads have invoked `board` before the captain invokes `rowBoat`.

`iAmTheCaptain` is a boolean, meaning it is true or false. Initially all threads set `iAmTheCaptain = false`. Only one thread from each boat sets it to `true`.

5.2.2 River crossing solution

The basic idea of this solution is that each arrival updates one of the counters and then checks whether it makes a full complement, either by being the fourth of its kind or by completing a mixed pair of pairs.

I'll present the code for hackers; the serf code is symmetric (except, of course, that it is 1000 times bigger, full of bugs, and it contains an embedded web browser):

Listing 5.5: River crossing solution

```
mutex.wait ()
    hackers++
    if (hackers == 4)
        hackerSem.signal (4)        // signal four times
        hackers = 0
        iAmTheCaptain = true
    else if (hackers == 2 and serfs >= 2)
        hackerSem.signal (2)            // signal twice
        serfSem.signal (2)              // signal twice
        serfs -= 2
        hackers = 0
        iAmTheCaptain = true
    else
        mutex.signal ()  // captain keeps the mutex

wait (hackerSem)        // wait here to board

board ()
barrier.wait ()         // wait until all 4 have boarded

if (iAmTheCaptain)
    mutex.signal ()
    rowBoat ()
```

This is the first case we have seen where a mutex does not have a clear end point. In most cases, threads enter the mutex, update the counter, and exit the mutex. Then they wait on `hackerSem` or `serfSem`.

But when a thread arrives that forms a complete boatload, it has to signal `hackerSem` and `serfSem` to allow the other threads to board. Then it adjusts the counters. Finally, it has to hold onto the mutex until all the threads have boarded.

The barrier keeps track of how many threads have boarded. When the last thread arrives, all threads proceed. One of them, the captain, releases the mutex (finally) and then rows the boat.

Here are some middle-level assertions you might want to prove to yourself:

- There can't be more than 4 hackers or 4 serfs in the boarding area at a time.
- When all four passengers have boarded, and the captain is about to invoke `rowBoat` there can be at most 1 thread in the boarding area.

5.3 The unisex bathroom problem

I wrote this problem[1] when my friend Shelby Nelson left her position teaching physics at Colby College and took a job at Xerox.

She was working in a cubicle in the basement of a concrete monolith, and the nearest women's bathroom was two floors up. She proposed to the Uberboss that they convert the men's bathroom on her floor to a unisex bathroom, sort of like on Ally McBeal.

The Uberboss agreed, provided that the following synchronization constraints can be maintained:

- There cannot be men and women in the bathroom at the same time.
- There should never be more than three employees squandering company time in the bathroom.

Of course the solution should avoid deadlock. For now, though, don't worry about starvation. You may assume that the bathroom is equipped with all the semaphores you need.

[1] Later I learned that a nearly identical problem appears in Andrews's *Concurrent Programming*[1]

5.3.1 Unisex bathroom hint

Here are the variables I used in my solution:

Listing 5.6: Unisex bathroom hint

```
Semaphore empty = 1
Louie maleLouie, femaleLouie
Semaphore maleMultiplex = 3
Semaphore femaleMultiplex = 3
```

`empty` is 1 if the room is empty and 0 otherwise.

`maleLouie` allows men to bar women from the room. When the first male enters, the filo locks `empty`, barring women; When the last male exist, it unlocks `empty`, allowing women to enter. Women do likewise using `femaleLouie`.

`maleMultiplex` and `femaleMultiplex` ensure that there are no more than three men and three women in the system at a time.

5.3.2 Unisex bathroom solution

Here is the female code:

Listing 5.7: Unisex bathroom solution (female)

```
femaleLouie.lock (empty)
   femaleMultiplex.wait ()
      bathroom code here
   femaleMultiplex.signal ()
female Louie.unlock (empty)
```

The male code is similar.

The exclusion pattern in this problem 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.

Are there any problems with this solution?

5.3.3 No-starve unisex bathroom problem

The problem with the previous solution is that it allows starvation. A long line of women can arrive and enter while there is a man waiting, and vice versa.

Puzzle: fix the problem.

5.3.4 No-starve unisex bathroom solution

As we have seen before, we can use a turnstile to allow one kind of thread to stop the flow of the other kind of thread. This time we'll look at the male code:

Listing 5.8: No-starve unisex bathroom solution (male)

```
turnstile.wait ()
   maleLouie.lock (empty)
turnstile.signal ()

   maleMultiplex.wait ()
      bathroom code here
   maleMultiplex.signal ()

maleLouie.unlock (empty)
```

As long as there are men in the room, new arrivals will pass through the turnstile and enter. If there are women in the room when a male arrives, the male will block inside the turnstile, which will bar all later arrivals (male and female) from entering until the current occupants leave. At that point the male in the turnstile enters, possibly allowing additional males to enter.

The female code is similar, so if there are men in the room an arriving female will get stuck in the turnstile, barring additional men.

5.3.5 Efficient no-starve unisex bathroom problem

There is one aspect of this solution that is possibly inefficient. If the system is busy, then there will often be several threads, male and female, queued on the turnstile. Each time `empty` is signalled, one thread will leave the turnstile and another will enter. If the new thread is the opposite gender, it will promptly block, barring additional threads. Thus, there will usually be only 1-2 threads in the bathroom at a line, and the system will not take advantage of the available parallelism.

Puzzle: Extend this solution so that when `empty` is signalled by an exiting thread, a waiting thread of the opposite gender proceeds (if there is one), along with up to two additional threads of the same gender.

5.3.6 Efficient no-starve unisex bathroom hint

The fundamental problem is that only one thread can wait inside the turnstile, so when the bathroom is empty we are only guaranteed that one new thread can enter.

We would like to replace the turnstile with a "waiting room," that can hold multiple threads of the same gender. The requirements for the waiting room are similar to the requirements for the bathroom itself – it has limited capacity and should enforce categorical mutual exclusion.

My solution uses the following additional variables.

Listing 5.9: Unisex bathroom hint

```
Semaphore waitRoomEmpty = 1
Louie maleWaitLouie, femaleWaitLouie
Semaphore maleWaitMultiplex = 3
Semaphore femaleWaitMultiplex = 3
```

5.3.7 Efficient unisex bathroom solution

Listing 5.10: Efficient unisex bathroom solution

```

maleWaitMultiplex.wait ()
    maleWaitLouie.lock (waitRoomEmpty)
        maleLouie.lock (empty)
    maleWaitLouie.unlock (waitRoomEmpty)
maleWaitMultiplex.signal ()

    maleMultiplex.wait ()
        bathroom code here
    maleMultiplex.signal ()

maleLouie.unlock (empty)
```

The multiplex allows up to three male threads to be in the waiting room. The filo and `waitRoomEmpty` enforce categorical mutual exclusion.

Imagine that there are female threads in the bathroom. Additional female threads pass through the waiting room without blocking. When the first male thread arrives, it blocks on `maleLouie`. At this point, female threads hold `empty` but male threads hold `waitRoomEmpty`. Thus, additional female threads cannot enter the waiting room. Up to two additional male threads can enter the waiting room, where they will also block on `maleLouie`. Any additional male threads will block on `maleWaitMultiplex`.

Eventually the last female thread will signal `empty`. At that point, the male threads in `maleLouie` unblock and enter the bathroom. When the last male thread leaves the waiting room, it signals `waitRoomEmpty`. At this point a female thread may enter the waiting room, but there is no guarantee that a male threads doesn't get there first.

So this solution solved the efficiency problem, but it brought back the starvation problem. Fortunately, we can solve the starvation problem the same way we did before, by adding a turnstile:

Listing 5.11: Efficient unisex bathroom solution

```
maleWaitMultiplex.wait ()

turnstile.wait ()
    maleWaitLouie.lock (waitRoomEmpty)
turnstile.signal ()

        maleLouie.lock (empty)
    maleWaitLouie.unlock (waitRoomEmpty)
maleWaitMultiplex.signal ()

    maleMultiplex.wait ()
        bathroom code here
    maleMultiplex.signal ()

maleLouie.unlock (empty)
```

At this point the solution is complicated enough that it is hard to be sure that it is correct. To be honest, the effort to make the solution more efficient may have been misguided. The original (no-starve) solution is probably adequate if the number of threads competing for the bathroom is small. If that is the case, then the original solution is probably more efficient just because it spends less time fiddling with semaphores!

Also, keep in mind that for the vast majority of applications correctness is more important than efficiency. Write code that is provably correct.

5.4 Baboon crossing problem

This problem is adapted from Tanenbaum's *Operating Systems: Design and Implementation* [9]. There is a deep canyon somewhere in Kruger National Park, South Africa, and a single rope that spans the canyon. Baboons can cross the canyon by swinging hand-over-hand on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths. Furthermore, the rope is only strong enough to hold 5 baboons. If there are more baboons on the rope at the same time, it will break.

Assuming that we can teach the baboons to use semaphores, we would like to design a synchronization scheme with the following properties:

- Once a baboon has begun to cross, it is guaranteed to get to the other side without running into a baboon going the other way.
- There are never more than 5 baboons on the rope.
- A continuing stream of baboons crossing in one direction should not bar baboons going the other way indefinitely (no starvation).

I will not include a solution to this problem for reasons that should be clear.

5.5 The search-insert-delete problem

This one is from Andrews's *Concurrent Programming* [1].

> Three kinds of threads share access to a singly-linked list: searchers, inserters and deleters. Searchers merely examine the list; hence they can execute concurrently with each other. Inserters add new items to the end of the list; insertions must be mutually exclusive to preclude two interters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and deletion must also be mutually exclusive with searches and insertions.

Puzzle: write code for searchers, inserters and deleters that enforces this kind of three-way categorical mutual exclusion.

5.5.1 Search-Insert-Delete hint

Listing 5.12: Search-Insert-Delete hint

```
Semaphore insertMutex = 1
Semaphore noSearcher = 1
Semaphore noInserter = 1
Louie searchLouie = 0
Louie insertLouie = 0
```

`insertMutex` ensures that only one inserter is in its critical section at a time. `noSearcher` and `noInserter` indicate (surprise) that there are no searchers and no inserters in their critical sections; a deleter needs to hold both of these to enter.

`searchLouie` and `insertLouie` are used by searchers and inserters to exclude deleters.

5.5.2 Search-Insert-Delete solution

Here is my solution:

Listing 5.13: Search-Insert-Delete solution (searcher)

```
searchLouie.wait (noSearcher)
\\ critical section
searchLouie.signal (noSearcher)
```

The only thing a searcher needs to worry about is a deleter. The first searcher in takes `noSearcher`; the last one out releases it.

Listing 5.14: Search-Insert-Delete solution (inserter)

```
insertLouie.wait (noInserter)
insertMutex.wait ()
\\ critical section
insertMutex.signal ()
insertLouie.signal (noInserter)
```

Similarly, the first inserter takes `noInserter` and the last one out releases it. Since searchers and inserters compete for different semaphores, they can be in their critical section concurrently. But `insertMutex` ensures that only one inserter is in the room at a time.

Listing 5.15: Search-Insert-Delete solution (deleter)

```
noSearcher.wait ()
noInserter.wait ()
\\ critical section
noInserter.signal ()
noSearcher.signal ()
```

Since the deleter holds both `noSearcher` and `noInserter`, it is guaranteed exclusive access. Of course, any time we see a thread holding more than one semaphore, we need to check for deadlocks. By trying out a few scenarios, you should be able to convince yourself that this solution is deadlock free.

On the other hand, like many categorical exclusion problems, this one is prone to starvation. As we saw in the Readers-Writers problem, we can sometimes mitigate this problem by giving priority to one category of threads according to application-specific criteria. But in general it is difficult to write an efficient solution (one that allows the maximum degree of concurrency) that avoids starvation.

5.6 The dining savages problem

Another one from the excellent collection in Andrews's *Concurrent Programming* [1].

> A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and the waits until the cook has refilled the pot.

Any number of savage threads run the following code:

Listing 5.16: Unsynchronized savage code

```
while (1) {
    getServingFromPot ()
    eat ()
}
```

And one cook thread runs this code:

Listing 5.17: Unsynchronized cook code

```
while (1) {
    putServingsInPot (M)
}
```

Puzzle: Add synchronization code for the savages and the cook that prevents deadlock, and awakens the cook only when the pot is empty.

Note: This problem is based on a cartoonish representation of the history of Western missionaries among hunter-gatherer societies. Some humor is intended by the allusion to the Dining Philosophers problem, but the representation of "savages" here isn't intended to be any more realistic than the previous representation of philosophers. If you are interested in hunter-gatherer societies, I recommend Jared Diamond's *Guns, Germs and Steel*, among many other popular books about anthropology.

5.6.1 Dining Savages hint

It is tempting to use a semaphore to keep track of the number of servings, as in the producer-consumer problem. But in order to signal the cook when the pot is empty, a thread would have to know before decrementing the semaphore whether it would have to wait, and we just can't do that.

An alternative is to use a mutex-protected counter to keep track of the number of servings. If a savage finds the counter at zero, he wakes the cook and waits for a signal that the pot is full. Here are the variables I used:

Listing 5.18: Dining Savages hint

```
int servings
Semaphore mutex = 1
Semaphore cook = 0
Semaphore savage = 0
```

5.6.2 Dining Savages solution

The code for the cook is straightforward:

Listing 5.19: Dining Savages solution (cook)

```
while (1) {
    cook.wait ()
    putServingsInPot (M)
    savage.signal ()
}
```

The code for the savages is only a little more complicated. As each savage passes through the mutex, he decrements `servings`, in effect making a reservation for the meal he is about to get.

Listing 5.20: Dining Savages solution (savage)

```
while (1) {
    mutex.get ()
        if (servings == 0)
            cook.signal ()
            savage.wait ()
            servings = M
        servings--
    mutex.signal ()

    getServingFromPot ()
    eat ()
}
```

This solution is deadlock-free. The only opportunity for deadlock comes when the thread that holds `mutex` waits for `savage`. But since that thread has just signalled `cook`, it is guaranteed to wake when the cook signals `savage`.

It is not immediately obvious that the reservation system works, since it seems possible for a thread to decrement `servings`, release the mutex, and then find the pot empty by the time it invokes `getServingFromPot`. I will leave it to you to convince yourself that that is not possible.

5.7 The barbershop problem

The original barbershop problem was proposed by Dijkstra. A variation of it appears in Silberschatz and Galvin's *Operating Systems Concepts* [7].

> A barbershop consists of a waiting room with n chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

To make the problem a little more concrete, I added the following information:

- If a customer thread arrives when the shop is full, it can invoke `exit`. This operation allows the thread to skip to whatever it is supposed to do next, or terminate.
- When the barber invokes `cutHair` there should be exactly one thread invoking `getHairCut` concurrently.

5.7.1 Barbershop hint

Listing 5.21: Barbershop hint

```
int customers
Semaphore mutex = 1
Semaphore customer = 0
Semaphore barber = 0
```

`customers` counts the number of customers in the shop; it is protected by `mutex`.

The barber waits on `customer` until a customer enters the shop, then the customer waits on `barber` until the barber signals him to take a seat.

5.7.2 Barbershop solution

If there are n customers in the waiting room and one in the barber chair, then the shop is full and any customers that arrive immediately invoke `exit`.

Otherwise each customer signals the barber and waits for a signal from the barber. This is exactly the rendezvous pattern from Section 3.2.

Listing 5.22: Barbershop solution (customer)

```
mutex.wait ()
    if (customers == n+1)
        mutex.signal ()
        exit ()
    customers++
mutex.signal ()

customer.signal ()
barber.wait ()
getHairCut ()

mutex.wait ()
    customers--
mutex.signal ()
```

The barber runs in a loop. Each time a customer signals, the barber wakes, signals one customer, and gives one hair cut. If another customer arrives while the barber is busy, then on the next iteration the barber will pass the `barber` semaphore without sleeping.

Listing 5.23: Barbershop solution (barber)

```
customer.wait ()
barber.signal ()
cutHair ()
```

5.8 Hilzer's Barbershop problem

William Stallings [8] presents a more complicated version of the barbershop problem, which he attributes to Ralph Hilzer at the California State University at Chico.

> Our barbershop has three chairs, three barbers, and a waiting area that can accomodate four customers on a sofa and that has standing room for additional customers. Fire codes limit the total number of customers in the shop to 20.
>
> A customer will not enter the shop if it is filled to capacity with other customers. Once inside, the customer takes a seat on the sofa or stands if the sofa is filled. When a barber is free, the customer that has been on the sofa the longest is served and, if there are any standing customers, the one that has been in the shop the longest takes a seat on the sofa. When a customer's haircut is finished, any barber can accept payment, but because there is only one cash register, payment is accepted for one customer at a time. The barbers divide their time among cutting hair, accepting payment, and sleeping in their chair waiting for a customer.

Customers should invoke the following functions, in order: `enterShop`, `sitOnSofa`, `sitInBarberChair`, `pay`, `exitShop`.

Barbers invoke `cutHair` and `acceptPayment`. The following synchronization constraints apply:

- Customers cannot invoke `enterShop` if the shop is at capacity.
- If the sofa is full, an arriving customer cannot invoke `sitOnSofa` until one of the customers on the sofa invokes `sitInBarberChair`.
- If all three barber chairs are busy, an arriving customer cannot invoke `sitInBarberChair` until one of the customers in a chair invokes `pay`.
- The customer has to `pay` before the barber can `acceptPayment`.
- The barber must `acceptPayment` before the customer can `exitShop`.

One difficulty with this problem is that in each waiting area (the sofa and the standing room), customers have to be served in first-in-first-out (FIFO) order. If our implementation of semaphores happens to enforce FIFO queueing, then we can use nested multiplexes to create the waiting areas. Otherwise we can use Fifo objects.

Puzzle: Write synchonization code that enforces all the synchronization constraints for Hilzer's barbershop.

5.8.1 Hilzer's barbershop hint

Here are the variables I used in my solution:

Listing 5.24: Hilzer's barbershop hint

```
Semaphore mutex = 1
int customers
Fifo standingRoom (16)
Fifo sofa (4)
Semaphore chair = 3
Semaphore barber = 0
Semaphore customer = 0
Semaphore cash = 0
Semaphore receipt = 0
```

`mutex` protects `customers`, which keeps track of the number of customers in the shop so that if a thread arrives when the shop is full, it can exit. `standingRoom` and `sofa` are Fifos that represent the waiting areas. `chair` is a multiplex that limits the number of customers in the seating area.

The other semaphores are used for the rendezvous between the barber and the customers. The customer signals `barber` and then wait on `customer`. Then the customer signals `cash` and waits on `receipt`.

5.8.2 Hilzer's barbershop solution

There is nothing here that is new; it's just a combination of patterns we have seen before.

Listing 5.25: Hilzer's barbershop solution (customer)

```
mutex.wait ()
    if (customers == 20)
        mutex.signal ()
        exit ()
    customers++
mutex.signal ()

standingRoom.wait ()
enterShop ()

sofa.wait()
standingRoom.signal ()
sitOnSofa ()

chair.wait ()
sofa.signal ()
sitInBarberChair ()

```

```
customer.signal ()
barber.wait ()
getHairCut ()

pay ()
cash.signal ()
receipt.wait ()

mutex.wait ()
    customers--
mutex.signal ()
leaveBarberShop ()
```

The mechanism that counts the number of customers and allows threads to exit is the same as in the previous problem.

In order to maintain FIFO order for the whole system, threads cascade from `standingRoom` to `sofa` to `chair` in much the same way they cascade from one semaphore to the next in the implementation of Fifo.

The exchange with the barber is basically two consecutive rendezvouses[2]

The code for the barber is self-explanatory:

Listing 5.26: Hilzer's barbershop solution (barber)

```
customer.wait ()
barber.signal ()
cutHair()

cash.wait ()
acceptPayment ()
receipt.signal ()
```

If two customers signal `cash` concurrently, there is no way to know which waiting barber will get which signal, but the problem specification says that's ok. Any barber can take money from any customer.

[2] The plural of rendezvous is rare, and not all dictionaries agree about what it is. Another possibility is that the plural is also spelled "rendezvous," but the final "s" is pronounced.

5.9 The room party problem

I wrote this problem while I was at Colby College. One semester there was a controversy over an allegation by a student that someone from the Dean of Students Office had searched his room in his absence. Although the allegation was public, the Dean of Students wasn't able to comment on the case, so we never found our what really happened. I wrote this problem to tease a friend of mine, who was the Dean of Student Housing.

The following synchronization constraints apply to students and the Dean of Students:

1. Any number of students can be in a room at the same time.
2. The Dean of Students can only enter a room if there are no students in the room (to conduct a search) or if there are more than 50 students in the room (to break up the party).
3. While the Dean of Students is in the room, no additional students may enter, but students may leave.
4. The Dean of Students may not leave the room until all students have left.
5. There is only one Dean of Students, so you do not have to enforce exclusion among multiple deans.

Puzzle: write synchronization code for students and for the Dean of Students that enforces all of these constraints.

5.9.1 Room party hint

Listing 5.27: Room party hint

```
int students                 // number of students in room
Semaphore mutex = 1          // protect the student count
Semaphore noDean = 1         // the dean is not in the room
Semaphore deanCanEnter = 1   // the dean may enter
Semaphore noStudents = 1     // there are no students in the room
```

`students` counts the number of students in the room. `mutex` protects the counter (and also allows the dean to bar students from entering).

`noDean` is 1 when the dean is not in the room. `noStudents` is 1 when there are no students in the room. `deanCanEnter` is 1 when there are no students or more than 50.

5.9.2 Room party solution

This problem is hard. I went through a lot of trial and error before I found a solution that (I am pretty sure) works. One of the difficulties is that we need to allow students to leave without allowing new students to enter; the solution I chose requires students to hold two mutexes (`mutex` and `noDean`) to enter and only one to leave.

`noDean` is a mutex for both students and deans. It prevents students from entering when the dean is in the room. It also prevents the dean from entering while a student is in the middle of checking in. Incidentally, it enforces mutual exclusion among multiple deans, which is not required, but it is acceptable.

`noStudents` is 1 if there are no students in the room and 0 otherwise. The student threads maintain it using a pattern similar to a Louie; the first student in locks it and the last one out unlocks it. The deans use `noStudents` to ensure that all the students have left the room before they exit.

`deanCanEnter` is a turnstile for deans. When there are no students in the room, the turnstile is unlocked. The first student to enter locks it, but the 50th unlocks it again. Similarly, when the number of students drops below 50, `deanCanEnter` gets locked again. When the last student exits, it is unlocked one more time.

Listing 5.28: Room party solution (student)

```
noDean.wait ()
mutex.wait ()
    count++
    if (count == 1)
        deanCanEnter.wait ()
        noStudents.wait ()
    if (count == 51)
        deanCanEnter.signal ()
mutex.signal ()
noDean.signal ()

party ()

mutex.wait ()
    count--
    if (count == 0)
        noStudents.signal ()
        deanCanEnter.signal ()
    if (count == 50)
        deanCanEnter.wait ()
mutex.signal ()
```

The dean's code uses `mutex` in a non-standard way. The dean gets the mutex to prevent students from changing the counter while the dean is checking it. If there are more than 50 students, then the dean has to release the mutex, allowing students to exit, before waiting on `noStudents`. Otherwise the program deadlocks.

Listing 5.29: Room party solution (dean)

```
deanCanEnter.wait ()
deanCanEnter.signal ()

noDean.wait ()
    mutex.wait ()
    if (count == 0)
        search ()
        mutex.signal ()
    if (count > 50)
        breakUpParty ()
        mutex.signal ()
        noStudents.wait ()
        noStudents.signal ()
    else
        mutex.signal ()
noDean.signal ()
```

In this solution, it is possible for students to enter and leave after the dean passes the turnstile, which means that by the time the dean checks the counter there might be more than 0 and fewer than 50 students in the room. In this case, the dean should exit without invoking `search` or `breakUpParty`. It is not efficient to allow deans to enter and exit without doing useful work, but we expect this to happen only occasionally, since it depends on a fluke of timing.

5.10 The Santa Claus problem

This problem is from William Stallings's *Operating Systems* [8], but he attributes it to John Trono of St. Michael's College in Vermont.

> Stand Claus sleeps in his shop at the North Pole and can only be awakened by either (1) all nine reindeer being back from their vacation in the South Pacific, or (2) some of the elves having difficulty making toys; to allow Santa to get some sleep, the elves can only wake him when three of them have problems. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, becuase it is more important to get his sleigh ready. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment.) The last reindeer to arrive must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Here are some addition specifications:

- After the ninth reindeer arrives, Santa must invoke `prepareSleigh`, and then all nine reindeer must invoke `getHitched`.
- After the third elf arrives, Santa must invoke `helpElves`. Concurrently, all three elves should invoke `getHelp`.
- All three elves must invoke `getHelp` before any additional elves enter (increment the elf counter).

Santa should run in a loop so he can help many sets of elves. We can assume that there are exactly 9 reindeer, but there may be any number of elves.

5.10.1 Santa problem hint

Listing 5.30: Santa problem hint

```
int elves = 0
int reindeer = 0
Semaphore santaSem = 0
Semaphore reindeerSem = 0
Semaphore elfTex = 1
Semaphore mutex = 1
```

`elves` and `reindeer` are counters, both protected by mutex. Elves and reindeer get `mutex` to modify the counters; Santa gets it to check them.

Santa waits on `santaSem` until either an elf or a reindeer signals him.

The reindeer wait on `reindeerSem` until Santa signals them to enter the paddock and get hitched.

The elves use `elfTex` to prevent additional elves from entering while three elves are being helped.

5.10.2 Santa problem solution

Santa's code is pretty straightforward. Remember that it runs in a loop.

Listing 5.31: Santa problem solution (Santa)

```
santaSem.wait ()
mutex.wait ()
    if (reindeer == 9)
        prepareSleigh ()
        reindeerSem.signal (9)
    else if (elves == 3)
        helpElves ()
mutex.signal ()
```

When Santa wakes up, he checks which of the two conditions holds and either deals with the reindeer or the waiting elves. If there are nine reindeer waiting, Santa invokes `prepareSleigh`, then signals `reindeerSem` nine times, allowing the reindeer to invoke `getHitched`. If there are elves waiting, Santa just invokes `helpElves`. There is no need for the elves to wait for Santa; once they signal `santaSem`, they can invoke `getHelp` immediately.

Here is the code for reindeer:

Listing 5.32: Santa problem solution (reindeer)

```
mutex.wait ()
    reindeer++
    if (reindeer == 9)
        santaSem.signal ()
mutex.signal ()

reindeerSem.wait ()
getHitched ()
```

The ninth reindeer signals Santa and then joins the other reindeer waiting on `reindeerSem`. When Santa signals, the reindeer all execute `getHitched`.

The elf code is similar, except that when the third elf arrives it has to bar subsequent arrivals until the first three have executed `getHelp`.

Listing 5.33: Santa problem solution (elves)

```
elfTex.wait ()
mutex.wait ()
    elves++
    if (elves == 3)
        santaSem.signal ()
    else
        elfTex.signal ()
mutex.signal ()

getHelp ()

mutex.wait ()
    elves--
    if (elves == 0)
       elfTex.signal ()
mutex.signal ()
```

The first two elves release `elfTex` at the same time they release the `mutex`, but the last elf holds `elfTex`, barring other elves from entering until all three elves have invoked `getHelp`.

The last elf to leave releases `elfTex`, allowing the next batch of elves to enter.

5.11 The roller coaster problem

This problem is from Andrews's *Concurrent Programming* [1], but he attibutes it to J. S. Herman's Master's thesis.

> Suppose there are n passenger threads and a car thread. The passengers repeatedly wait to take rides in the car, which can hold C passengers, where $C < n$. The car can go around the tracks only when it is full.

Here are some additional details:

- After the car arrives, C passengers should invoke `boardCar`, then the car should invoke `depart`, then passengers should invoke `leaveCar`.

Puzzle: Write code for the passengers and car that enforces these constraints.

5.11.1 Roller Coaster hint

Listing 5.34: Roller Coaster hint

```
Semaphore car = 0
Semaphore allAboard = 0
Semaphore lastStop = 0
Semaphore mutex = 1
int passengers = 0
```

`car` indicates that the car has arrived; `allAboard` indicates that the car is full; `lastStop` indicates that the car has departed and looped the track.

`mutex` protects `passengers`, which counts the number of passengers that have invoked `boardCar`.

5.11.2 Roller Coaster solution

Listing 5.35: Roller Coaster solution (car)

```
car.signal (C)
allAboard.wait ()
depart ()
lastStop.signal (C)
```

When the car arrives, it allows C passengers to proceed, then waits for the last one to invoke `boardCar`. When it has departed, it allows C passengers to disembark.

Listing 5.36: Roller Coaster solution (passenger)

```
car.wait ()
boardCar ()

mutex.wait ()
   passengers++
   if (passengers == C)
       allAboard.signal ()
       passengers = 0
mutex.signal ()

lastStop.wait ()
leaveCar ()
```

Passengers wait for the car before boarding, naturally, and wait for the car to stop before leaving. The last passenger to board signals the car and resets the passenger counter.

This solution does not generalize to the case where there is more than one car. In order to do that, we have to satisfy some additional constraints:

- Only one car can be boarding at a time.
- Multiple cars can be on the track concurrently.
- Since cars can't pass each other, they have to unload in the same order they boarded.

Puzzle: assuming that we know that there are m cars, modify the previous solution to handle the additional constraints.

5.11.3 Multi-car Roller Coaster hint

We can assume that each car has a local variable `i` that contains an identifier between 0 and $m-1$. One way to initialize these variables is to pass the thread through a barrier that assigns identifiers in order.

Two arrays of m semaphores keep the cars in order. As each thread enters the boarding area, or prepares to unload, it waits on its own semaphore, and then signals the next car in line.

Listing 5.37: Multi-car Roller Coaster hint

```
int m
local int i
Semaphore board[m] = { 1, 0, 0, ..., 0 }
Semaphore unload[m] = { 1, 0, 0, ..., 0 }
```

The function `next` computes the identifier of the next car in the sequence (wrapping around from $m-1$ to 0):

Listing 5.38: Implementation of `next`

```
int next (i) { return (i + 1) mod m }
```

5.11.4 Multi-car Roller Coaster solution

Initially, only the car with identifier 0 can proceed. When each car finishes boarding, it signals its successor. Any number of cars can invoke `depart` concurrently, but when they enter the unloading area, again, they have to go in order.

When the last car leaves an area, it signals the first, which restores the original state of the semaphore array (or allows the first car to proceed, if it is already waiting).

Listing 5.39: Multi-car Roller Coaster solution (car)

```
board[i].wait()
    car.signal (C)
    allAboard.wait ()
board[next(i)].signal()

depart ()

unload[i].wait()
    lastStop.signal (C)
unload[next(i)].signal ()
```

In my first draft of this solution, I used the same array of semaphores for `board` and `unload`. Why doesn't that work?

One potential problem with this solution is that a car can start loading again before the passengers from the previous load have left. If that bothers you, you should have no trouble fixing it.

5.12 The Bus problem

This problem was originally based on the Senate bus at Wellesley College. Riders come to a bus stop and wait for a bus. When the bus arrives, all the waiting riders invoke `boardBus`, but anyone who arrives while the bus is boarding has to wait for the next bus. The capacity of the bus is 50 people; if there are more than 50 people waiting, some will have to wait for the next bus.

When all the waiting riders have boarded, the bus can invoke `depart`. If the bus arrives when there are no riders, it should depart immediately.

Puzzle: Write synchronization code that enforces all of these constraints.

5.12.1 Bus problem hint

Here are the variables I used in my solution:

Listing 5.40: Bus problem hint

```
Semaphore mutex = 1;
int riders;
Semaphore turnstile = 1;
Semaphore multiplex = 50;
Semaphore bus = 0;
Semaphore allAboard = 0;
```

`mutex` protects `riders`, which keeps track of how many riders are waiting. The `turnstile` controls access to the waiting area; it is initially open. When the bus arrives, it locks the turnstile until everyone has boarded, which bars late arrivals. The `multiplex` makes sure there are no more than 50 riders in the boarding area.

Riders wait on `bus`, which gets signalled when the bus arrives. The bus waits on `allAboard`, which gets signalled by the last student to board.

5.12.2 Bus problem solution

Here is the code for the bus:

Listing 5.41: Bus problem solution (bus)

```
turnstile.wait()
    if (riders == 0)
        depart()
        turnstile.signal
        return

bus.signal()
allAboard.wait()

bus.wait()
depart()
turnstile.signal()
```

When the bus arrives, it gets and keeps `turnstile`, which prevents late arrivals from entering the boarding area. If there are no rides, it departs immediately. Otherwise, it signals `bus` and waits for the riders to board.

Here is the code for the riders:

Listing 5.42: Bus problem solution (riders)

```
multiplex.wait()

    turnstile.wait()
        riders++
    turnsile.signal()

    bus.wait()
    bus.signal()
multiplex.signal()
boardBus()

mutex.wait ()
    riders--
    if (riders == 0) allAboard.signal();
mutex.signal ()
```

The multiplex controls the number of riders in the waiting area, although strictly speaking, a rider doesn't enter the waiting area until she passes the turnstile. We increment `riders` inside the turnstile to guarantee that the count is accurate when the bus arrives.

Riders wait on `bus` and then pass through one at a time. After boarding, each rider gets the mutex and decrements the counter. When the last thread boards, it signals `allAboard`, which allows the bus to leave.

For the most part, this solution is just a combination of patterns we have seen before. But there is one subtlety: notice that `riders` is not always protected by the same mutex. The increment (Line 4) is protected by `turnstile` but the decrement (Line 13) is protected by `mutex`. So we have to check whether one thread might be incrementing while another is decrementing. Fortunately, we can prove that this is impossible. If a rider holds `mutex`, then the bus must be boarding, in which case the bus holds `turnstile`, so no threads can be incrementing riders.

Challenge: if riders arrive while the bus is boarding, they might be annoyed if you make them wait for the next one. Can you find a solution that allows late arrivals to board without violating the other constraints?

Bibliography

[1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

[2] Edsgar Dijkstra. Cooperating sequential processes. 1965. Reprinted in *Programming Languages*, F. Genuys, ed., Academic Press, New York 1968.

[3] Armando R. Gingras. Dining philosophers revisited. *ACM SIGCSE Bulletin*, 22(3):21–24, 28, September 1990.

[4] Joseph M. Morris. A starvation-free solution to the mutual exclusion problem. *Information Processing Letters*, 8:76–80, February 1979.

[5] David L. Parnas. On a solution to the cigarette smokers' problem without conditional statements. *Communications of the ACM*, 18:181–183, March 1975.

[6] Suhas Patil. Limitations and capabilities of dijkstra's semaphore primitives for coordination among processes. Technical report, MIT, 1971.

[7] Abraham Silberschatz and Peter Baer Galvin. *Operating Systems Concepts*. Addison Wesley, fifth edition, 1997.

[8] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, fourth edition, 2000.

[9] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.