Chapter 5

Less classical synchronization problems

5.1 The dining savages problem

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

> A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary[1]. 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 then waits until the cook has refilled the pot.

Any number of savage threads run the following code:

Unsynchronized savage code

```
while True:
    getServingFromPot()
    eat()
```

And one cook thread runs this code:

Unsynchronized cook code

```
while True:
    putServingsInPot(M)
```

[1] 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*, Napoleon Chagnon's *The Yanomamo*, and Redmond O'Hanlon's *In Trouble Again*, but not Tierney's *Darkness in El Dorado*, which I believe is unreliable.

The synchronization constraints are:

- Savages cannot invoke `getServingFromPot` if the pot is empty.
- The cook can invoke `putServingsInPot` only if the pot is empty.

Puzzle: Add code for the savages and the cook that satisfies the synchronization constraints.

5.1.1 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 scoreboard 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:

Dining Savages hint

```
servings = 0
mutex = Semaphore(1)
emptyPot = Semaphore(0)
fullPot = Semaphore(0)
```

Not surprisingly, `emptyPot` indicates that the pot is empty and `fullPot` indicates that the pot is full.

5.1.2 Dining Savages solution

My solution is a combination of the scoreboard pattern with a rendezvous. Here is the code for the cook:

Dining Savages solution (cook)

```
while True:
    emptyPot.wait()
    putServingsInPot(M)
    fullPot.signal()
```

The code for the savages is only a little more complicated. As each savage passes through the mutex, he checks the pot. If it is empty, he signals the cook and waits. Otherwise, he decrements `servings` and gets a serving from the pot.

Dining Savages solution (savage)

```
while True:
    mutex.wait()
        if servings == 0:
            emptyPot.signal()
            fullPot.wait()
            servings = M
        servings -= 1
        getServingFromPot()
    mutex.signal()

    eat()
```

It might seem odd that the savage, rather than the cook, sets `servings = M`. That's not really necessary; when the cook runs `putServingsInPot`, we know that the savage that holds the mutex is waiting on `fullPot`. So the cook could access `servings` safely. But in this case, I decided to let the savage do it so that it is clear from looking at the code that all accesses to `servings` are inside the mutex.

This solution is deadlock-free. The only opportunity for deadlock comes when the savage that holds `mutex` waits for `fullPot`. While he is waiting, other savages are queued on `mutex`. But eventually the cook will run and signal `fullPot`, which allows the waiting savage to resume and release the mutex.

Does this solution assume that the pot is thread-safe, or does it guarantee that `putServingsInPot` and `getServingFromPot` are executed exclusively?

5.2 The barbershop problem

The original barbershop problem was proposed by Dijkstra. A variation of it appears in Silberschatz and Galvin's *Operating Systems Concepts* [10].

> 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:

- Customer threads should invoke a function named `getHairCut`.
- If a customer thread arrives when the shop is full, it can invoke `balk`, which does not return.
- The barber thread should invoke `cutHair`.
- When the barber invokes `cutHair` there should be exactly one thread invoking `getHairCut` concurrently.

Write a solution that guarantees these constraints.

5.2.1 Barbershop hint

Barbershop hint

```
n = 4
customers = 0
mutex = Semaphore(1)
customer = Semaphore(0)
barber = Semaphore(0)
customerDone = Semaphore(0)
barberDone = Semaphore(0)
```

`n` is the total number of customers that can be in the shop: three in the waiting room and one in the chair.

`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.

After the haircut, the customer signals `customerDone` and waits on `barberDone`.

5.2.2 Barbershop solution

This solution combines a scoreboard and two rendezvouses[2]. Here is the code for customers:

Barbershop solution (customer)

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

customer.signal()
barber.wait()

# getHairCut()

customerDone.signal()
barberDone.wait()

mutex.wait()
    customers -= 1
mutex.signal()
```

If there are n customers in the shop any customers that arrive immediately invoke `balk`. Otherwise each customer signals `customer` and waits on `barber`.

Here is the code for barbers:

Barbershop solution (barber)

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

# cutHair()

customerDone.wait()
barberDone.signal()
```

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

The names for `customer` and `barber` are based on the naming convention for a rendezvous, so `customer.wait()` means "wait for a customer," not "customers wait here."

[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.

The second rendezvous, using `customerDone` and `barberDone`, ensures that the hair cut is done before the barber loops around to let the next customer into the critical section.

This solution is in `sync_code/barber.py` (see 3.2).

5.3 The FIFO barbershop

In the previous solution there is no guarantee that customers are served in the order they arrive. Up to `n` customers can pass the turnstile, signal `customer` and wait on `barber`. When the barber signal `barber`, any of the customers might proceed.

Modify this solution so that customers are served in the order they pass the turnstile.

Hint: you can refer to the current thread as `self`, so if you write `self.sem = Semaphore(0)`, each thread gets its own semaphore.

5.3.1 FIFO barbershop hint

My solution uses a list of semaphores named `queue`.

FIFO barbershop hint

```
n = 4
customers = 0
mutex = Semaphore(1)
customer = Semaphore(0)
customerDone = Semaphore(0)
barberDone = Semaphore(0)
queue = []
```

As each thread passes the turnstile, it creates a thread and puts it in the queue.

Instead of waiting on `barber`, each thread waits on its own semaphore. When the barber wakes up, he removes a thread from the queue and signals it.

5.3.2 FIFO barbershop solution

Here is the modified code for customers:

FIFO barbershop solution (customer)

```
self.sem = Semaphore(0)
mutex.wait()
    if customers == n:
        mutex.signal()
        balk()
    customers += 1
    queue.append(self.sem)
mutex.signal()

customer.signal()
self.sem.wait()

# getHairCut()

customerDone.signal()
barberDone.wait()

mutex.wait()
    customers -= 1
mutex.signal()
```

And the code for barbers:

FIFO barbershop solution (barber)

```
customer.wait()
mutex.wait()
    sem = queue.pop(0)
mutex.signal()

sem.signal()

# cutHair()

customerDone.wait()
barberDone.signal()
```

Notice that the barber has to get `mutex` to access the queue.
This solution is in `sync_code/barber2.py` (see 3.2).

5.4 Hilzer's Barbershop problem

William Stallings [11] 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 accommodate 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.

In other words, the following synchronization constraints apply:

- Customers invoke the following functions in order: `enterShop`, `sitOnSofa`, `getHairCut`, `pay`.
- Barbers invoke `cutHair` and `acceptPayment`.
- Customers cannot invoke `enterShop` if the shop is at capacity.
- If the sofa is full, an arriving customer cannot invoke `sitOnSofa`.
- When a customer invokes `getHairCut` there should be a corresponding barber executing `cutHair` concurrently, and vice versa.
- It should be possible for up to three customers to execute `getHairCut` concurrently, and up to three barbers to execute `cutHair` concurrently.
- The customer has to `pay` before the barber can `acceptPayment`.
- The barber must `acceptPayment` before the customer can exit.

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

5.4.1 Hilzer's barbershop hint

Here are the variables I used in my solution:

Hilzer's barbershop hint

```
n = 20
customers = 0
mutex = Semaphore(1)
sofa = Semaphore(4)
customer1 = Semaphore(0)
customer2 = Semaphore(0)
barber = Semaphore(0)
payment = Semaphore(0)
receipt = Semaphore(0)
queue1 = []
queue2 = []
```

`mutex` protects `customers`, which keeps track of the number of customers in the shop, and `queue1` which is a list of semaphores for threads waiting for a seat on the sofa.

`mutex2` protects `queue2`, which is a list of semaphores for threads waiting for a chair.

`sofa` is a multiplex that enforces the maximum number of customers on the sofa.

`customer1` signals that there is a customer in `queue1`, and `customer2` signals that there is a customer in `queue2`.

`payment` signals that a customer has paid, and `receipt` sigmals that a barber has accepted payment.

5.4.2 Hilzer's barbershop solution

This solution is considerably more complex than I expected. I am not if Hilzer had something simpler in mind, but here is the best I could do.

Hilzer's barbershop solution (customer)

```
self.sem1 = Semaphore(0)
self.sem2 = Semaphore(0)

mutex.wait()
    if customers == n:
        mutex.signal()
        balk()
    customers += 1
    queue1.append(self.sem1)
mutex.signal()

# enterShop()
customer1.signal()
self.sem1.wait()

sofa.wait()
    # sitOnSofa()
    self.sem1.signal()
    mutex.wait()
        queue2.append(self.sem2)
    mutex.signal()
    customer2.signal()
    self.sem2.wait()
sofa.signal()

# sitInBarberChair()

# pay()
payment.signal()
receipt.wait()

mutex.wait()
    customers -= 1
mutex.signal()
```

The first paragraph is the same as in the previous solution. When a customer arrives, it checks the counter and either balks or adds itself to the queue. Then it signals a barber.

When the customer gets out of queue, it enters the multiplex, sits on the couch and adds itself to the second queue.

When it gets out of *that* queue, it gets a haircut, pays, and then exits.

Hilzer's barbershop solution (barber)

```
customer1.wait()
mutex.wait()
    sem = queue1.pop(0)
    sem.signal()
    sem.wait()
mutex.signal()
sem.signal()

customer2.wait()
mutex.wait()
    sem = queue2.pop(0)
mutex.signal()
sem.signal()

barber.signal()
# cutHair()

payment.wait()
# acceptPayment()
receipt.signal()
```

Each barber waits for a customer to enter, signals the customer's semaphore to get it out of queue, then waits for it to claim a seat on the sofa. This enforces the FIFO requirement.

The barber waits for the customer to join the second queue and then signals it, allowing the customer to claim a chair.

Each barber admits one customer to the chair, so there can by up to three concurrent haircuts. Because there is only one cash register, the customer has to get `mutex`. The customer and barber rendezvous at the cash register, then both exit.

This solution satisfies the synchonization constraints, but it leaves the sofa underutilized. Because there are only three barbers, there can never be more than three customers on the sofa, so the multiplex is unnecessary.

This solution is in `sync_code/barber3.py` (see 3.2).

The only way I can think of to solve this problem is to create a third kind of thread, which I can an usher. The ushers manage `queue1` and the barbers manage `queue2`. If there are 4 ushers and 3 barbers, the sofa can be fully utilized.

This solution is in `sync_code/barber4.py` (see 3.2).

5.5 The Santa Claus problem

This problem is from William Stallings's *Operating Systems* [11], 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, because 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.

5.5.1 Santa problem hint

Santa problem hint

```
elves = 0
reindeer = 0
santaSem = Semaphore(0)
reindeerSem = Semaphore(0)
elfTex = Semaphore(1)
mutex = Semaphore(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.

5.5.2 Santa problem solution

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

Santa problem solution (Santa)

```
santaSem.wait()
mutex.wait()
    if reindeer >= 9:
        prepareSleigh()
        reindeerSem.signal(9)
        reindeer -= 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.

Santa doesn't have to decrement the `elves` counter because the elves do it on their way out.

Here is the code for reindeer:

Santa problem solution (reindeer)

```
mutex.wait()
    reindeer += 1
    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`.

Santa problem solution (elves)

```
elfTex.wait()
mutex.wait()
    elves += 1
    if elves == 3:
        santaSem.signal()
    else
        elfTex.signal()
mutex.signal()

getHelp()

mutex.wait()
    elves -= 1
    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.

5.6 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 `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.

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; thus, if we examine the sequence of threads that invoke `bond` and divide them into groups of three, each group should contain one oxygen and two hydrogen threads.

Puzzle: Write synchronization code for oxygen and hydrogen molecules that enforces these constraints.

5.6.1 H_2O hint

Here are the variables I used in my solution:

Water building hint

```
mutex = Semaphore(1)
oxygen = 0
hydrogen = 0
barrier = Barrier(3)
oxyQueue = Semaphore(0)
hydroQueue = Semaphore(0)
```

`oxygen` and `hydrogen` are counters, protected by `mutex`. `barrier` is where each set of three threads meets after invoking `bond` and before allowing the next set of threads to proceed.

`oxyQueue` is the semaphore oxygen threads wait on; `hydroQueue` is the semaphore hydrogen threads wait on. I am using the naming convention for queues, so `oxyQueue.wait()` means "join the oxygen queue" and `oxyQueue.signal()` means "release an oxygen thread from the queue."

5.6.2 H_2O solution

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

Oxygen code

```
mutex.wait()
oxygen += 1
if hydrogen >= 2:
    hydroQueue.signal(2)
    hydrogen -= 2
    oxyQueue.signal()
    oxygen -= 1
else:
    mutex.signal()

oxyQueue.wait()
bond()

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

As each oxygen thread enters, it gets the mutex and checks the scoreboard. If there are at least two hydrogen threads waiting, it signals two of them and itself and then bonds. If not, it releases the mutex and waits.

After bonding, threads wait at the barrier until all three threads have bonded, and then the oxygen thread releases the mutex. Since there is only one oxygen thread in each set of three, we are guaranteed to signal `mutex` once.

The code for hydrogen is similar:

Hydrogen code

```
mutex.wait()
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1:
    hydroQueue.signal(2)
    hydrogen -= 2
    oxyQueue.signal()
    oxygen -= 1
else:
    mutex.signal()

hydroQueue.wait()
bond()

barrier.wait()
```

An unusual feature of this solution is that the exit point of the mutex is ambiguous. In some cases, threads enter the mutex, update the counter, and exit the mutex. But when a thread arrives that forms a complete set, it has to keep the mutex in order to bar subsequent threads until the current set have invoked `bond`.

After invoking `bond`, the three threads wait at a barrier. When the barrier opens, we know that all three threads have invoked `bond` and that one of them holds the mutex. We don't know *which* thread holds the mutex, but it doesn't matter as long as only one of them releases it. Since we know there is only one oxygen thread, we make it do the work.

This might seem wrong, because until now it has generally been true that a thread has to hold a lock in order to release it. But there is no rule that says that has to be true. This is one of those cases where it can be misleading to think of a mutex as a token that threads acquire and release.

5.7 River crossing problem

This 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.

5.7.1 River crossing hint

Here are the variables I used in my solution:

River crossing hint

```
barrier = Barrier(4)
mutex = Semaphore(1)
hackers = 0
serfs = 0
hackerQueue = Semaphore(0)
serfQueue = Semaphore(0)
local isCaptain = 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. This is another example of a scoreboard.

`hackerQueue` and `serfQueue` 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`.

`isCaptain` is a local variable that indicates which thread should invoke `row`.

5.7.2 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):

River crossing solution

```
mutex.wait()
    hackers += 1
    if hackers == 4:
        hackerQueue.signal(4)
        hackers = 0
        isCaptain = True
    elif hackers == 2 and serfs >= 2:
        hackerQueue.signal(2)
        serfQueue.signal(2)
        serfs -= 2
        hackers = 0
        isCaptain = True
    else:
        mutex.signal()      # captain keeps the mutex

hackerQueue.wait()

board()
barrier.wait()

if isCaptain:
    rowBoat()
    mutex.signal()          # captain releases the mutex
```

As each thread files through the mutual exclusion section, it checks whether a complete crew is ready to board. If so, it signals the appropriate threads, declares itself captain, and holds the mutex in order to bar additional threads until the boat has sailed.

The barrier keeps track of how many threads have boarded. When the last thread arrives, all threads proceed. The captain invoked `row` and then (finally) releases the mutex.

5.8 The roller coaster problem

This problem is from Andrews's *Concurrent Programming* [1], but he attributes 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:

- Passengers should invoke `board` and `unboard`.
- The car should invoke `load`, `run` and `unload`.
- Passengers cannot board until the car has invoked `load`
- The car cannot depart until C passengers have boarded.
- Passengers cannot unboard until the car has invoked `unload`.

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

5.8.1 Roller Coaster hint

Roller Coaster hint

```
mutex = Semaphore(1)
mutex2 = Semaphore(1)
boarders = 0
unboarders = 0
boardQueue = Semaphore(0)
unboardQueue = Semaphore(0)
allAboard = Semaphore(0)
allAshore = Semaphore(0)
```

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

Passengers wait on `boardQueue` before boarding and `unboardQueue` before unboarding. `allAboard` indicates that the car is full.

5.8.2 Roller Coaster solution

Here is my code for the car thread:

Roller Coaster solution (car)

```
load()
boardQueue.signal(C)
allAboard.wait()

run()

unload()
unboardQueue.signal(C)
allAshore.wait()
```

When the car arrives, it signals C passengers, then waits for the last one to signal `allAboard`. After it departs, it allows C passengers to disembark, then waits for `allAshore`.

Roller Coaster solution (passenger)

```
boardQueue.wait()
board()

mutex.wait()
   boarders += 1
   if boarders == C:
       allAboard.signal()
       boarders = 0
mutex.signal()

unboardQueue.wait()
unboard()

mutex2.wait()
   unboarders += 1
   if unboarders == C:
       allAshore.signal()
       unboarders = 0
mutex2.signal()
```

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.

5.8.3 Multi-car Roller Coaster problem

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.
- All the threads from one carload must disembark before any of the threads from subsequent carloads.

Puzzle: modify the previous solution to handle the additional constraints. You can assume that there are m cars, and that each car has a local variable named `i` that contains an identifier between 0 and $m - 1$.

5.8.4 Multi-car Roller Coaster hint

I used two lists of semaphores to keep the cars in order. One represents the loading area and one represents the unloading area. Each list contains one semaphore for each car. At any time, only one semaphore in each list is unlocked, so that enforces the order threads can load and unload. Initially, only the semaphores for Car 0 are unlocked. As each car enters the loading (or unloading) it waits on its own semaphore; as it leaves it signals the next car in line.

Multi-car Roller Coaster hint

```
loadingArea = [Semaphore(0) for i in range(m)]
loadingArea[1].signal()
unloadingArea = [Semaphore(0) for i in range(m)]
unloadingArea[1].signal()
```

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

Implementation of `next`

```
def next(i):
    return (i + 1) % m
```

5.8.5 Multi-car Roller Coaster solution

Here is the modified code for the cars:

Multi-car Roller Coaster solution (car)

```
loadingArea[i].wait()
load()
boardQueue.signal(C)
allAboard.wait()
loadingArea[next(i)].signal()

run()

unloadingArea[i].wait()
unload()
unboardQueue.signal(C)
allAshore.wait()
unloadingArea[next(i)].signal()
```

The code for the passengers is unchanged.

Chapter 6

Not-so-classical problems

6.1 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 inserters 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.

6.1.1 Search-Insert-Delete hint

Search-Insert-Delete hint

```
insertMutex = Semaphore(1)
noSearcher = Semaphore(1)
noInserter = Semaphore(1)
searchSwitch = Lightswitch()
insertSwitch = Lightswitch()
```

`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.

`searchSwitch` and `insertSwitch` are used by searchers and inserters to exclude deleters.

6.1.2 Search-Insert-Delete solution

Here is my solution:

Search-Insert-Delete solution (searcher)

```
searchSwitch.wait(noSearcher)
# critical section
searchSwitch.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.

Search-Insert-Delete solution (inserter)

```
insertSwitch.wait(noInserter)
insertMutex.wait()
# critical section
insertMutex.signal()
insertSwitch.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.

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.

6.2 The unisex bathroom problem

I wrote this problem[1] when a friend of mine 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]

6.2.1 Unisex bathroom hint

Here are the variables I used in my solution:

Unisex bathroom hint

```
empty = Semaphore(1)
maleSwitch = Lightswitch()
femaleSwitch = Lightswitch()
maleMultiplex = Semaphore(3)
femaleMultiplex = Semaphore(3)
```

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

`maleSwitch` allows men to bar women from the room. When the first male enters, the lightswitch locks `empty`, barring women; When the last male exits, it unlocks `empty`, allowing women to enter. Women do likewise using `femaleSwitch`.

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

6.2.2 Unisex bathroom solution

Here is the female code:

Unisex bathroom solution (female)

```
femaleSwitch.lock(empty)
    femaleMultiplex.wait()
        # bathroom code here
    femaleMultiplex.signal()
female Switch.unlock(empty)
```

The male code is similar.
Are there any problems with this solution?

6.2.3 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.

6.2.4 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:

No-starve unisex bathroom solution (male)

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

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

maleSwitch.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.

This solution may not be efficient. If the system is busy, then there will often be several threads, male and female, queued on the turnstile. Each time `empty` is signaled, 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 time, and the system will not take full advantage of the available concurrency.

6.3 Baboon crossing problem

This problem is adapted from Tanenbaum's *Operating Systems: Design and Implementation* [12]. 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.

6.4 The Modus Hall Problem

This problem was written by Nathan Karst, one of the Olin students living in Modus Hall[2] during the winter of 2005.

> After a particularly heavy snowfall this winter, the denizens of Modus Hall created a trench-like path between their cardboard shantytown and the rest of campus. Every day some of the residents walk to and from class, food and civilization via the path; we will ignore the indolent students who chose daily to drive to Tier 3. We will also ignore the direction in which pedestrians are traveling. For some unknown reason, students living in West Hall would occasionally find it necessary to venture to the Mods.
>
> Unfortunately, the path is not wide enough to allow two people to walk side-by-side. If two Mods persons meet at some point on the path, one will gladly step aside into the neck high drift to accommodate the other. A similar situation will occur if two ResHall inhabitants cross paths. If a Mods heathen and a ResHall prude meet, however, a violent skirmish will ensue with the victors determined solely by strength of numbers; that is, the faction with the larger population will force the other to wait.

This is similar to the Baboon Crossing problem (in more ways than one), with the added twist that control of the critical section is determined by majority rule. This has the potential to be an efficient and starvation-free solution to the categorical exclusion problem.

Starvation is avoided because while one faction controls the critical section, members of the other faction accumulate in queue until they achieve a majority. Then they can bar new opponents from entering while they wait for the critical section to clear. I expect this solution to be efficient because it will tend to move threads through in batches, allowing maximum concurrency in the critical section.

Puzzle: write code that implements categorical exclusion with majority rule.

[2]Modus Hall is one of several nicknames for the modular buildings, aka Mods, that some students lived in while the second residence hall was being built.

6.4.1 Modus Hall problem hint

Here are the variables I used in my solution.

Modus problem hint

```
heathens = 0
prudes = 0
status = 'neutral'
mutex = Semaphore(1)
heathenTurn = Semaphore(1)
prudeTurn = Semaphore(1)
heathenQueue = Semaphore(0)
prudeQueue = Semaphore(0)
```

`heathens` and `prudes` are counters, and `status` records the status of the field, which can be 'neutral', 'heathens rule', 'prudes rule', 'transition to heathens' or 'transition to prudes'. All three are protected by `mutex` in the usual scoreboard pattern.

`heathenTurn` and `prudeTurn` control access to the field so that we can bar one side or the other during a transition.

`heathenQueue` and `prudeQueue` are where threads wait after checking in and before taking the field.

6.4.2 Modus Hall problem solution

Here is the code for heathens:

Modus problem solution

```
heathenTurn.wait()
heathenTurn.signal()

mutex.wait()
heathens++

if status == 'neutral':
    status = 'heathens rule'
    mutex.signal()
elif status == 'prudes rule':
    if heathens > prudes:
        status = 'transition to heathens'
        prudeTurn.wait()
    mutex.signal()
    heathenQueue.wait()
elif status == 'transition to heathens':
    mutex.signal()
    heathenQueue.wait()
else
    mutex.signal()

# cross the field

mutex.wait()
heathens--

if heathens == 0:
    if status == 'transition to prudes':
        prudeTurn.signal()
    if prudes:
        prudeQueue.signal(prudes)
        status = 'prudes rule'
    else:
        status = 'neutral'

if status == 'heathens rule':
    if prudes > heathens:
        status = 'transition to prudes'
        heathenTurn.wait()

mutex.signal()
```

As each student checks in, he has to consider the following cases:

- If the field is empty, the student lays claim for the heathens.
- If the heathens currently in charge, but the new arrival has tipped the balance, he locks the prude turnstile and the system switches to transition mode.
- If the prudes in charge, but the new arrival doesn't tip the balance, he joins the queue.
- If the system is transitioning to heathen control, the new arrival joins the queue.
- Otherwise we conclude that either the heathens are in charge, or the system is transitioning to prude control. In either case, this thread can proceed.

Similarly, as each student checks out, she has to consider several cases.

- If she is the last heathen to check out, she has to consider the following:
  - If the system is in transition, that means that the prude turnstile is locked, so she has to open it.
  - If there are prudes waiting, she signals them and updates `status` so the prudes are in charge. If not, the new status is 'neutral'.
- If she is not the last heathen to check out, she still has to check the possibility that her departure will tip the balance. In that case, she closes the heathen turnstile and starts the transition.

One potential difficulty of this solution is that any number of threads could be interrupted at Line 3, where they would have passed the turnstile but not yet checked in. Until they check in, they are not counted, so the balance of power may not reflect the number of threads that have passed the turnstile. Also, a transition ends when all the threads that have checked in have also checked out. At that point, there may be threads (of both types) that have passed the turnstile.

These behaviors may affect efficiency—this solution does not guarantee maximum concurrency—but they don't affect correctness, if you accept that "majority rule" only applies to threads that have registered to vote.

Chapter 7

Not remotely classical problems

7.1 The sushi bar problem

This problem was inspired by a problem proposed by Kenneth Reek [9]. Imagine a sushi bar with 5 seats. If you arrive while there is an empty seat, you can take a seat immediately. But if you arrive when all 5 seats are full, that means that all of them are dining together, and you will have to wait for the entire party to leave before you sit down.

Puzzle: write code for customers entering and leaving the sushi bar that enforces these requirements.

7.1.1 Sushi bar hint

Here are the variables I used:

Sushi bar hint

```
eating = waiting = 0
mutex = Semaphore(1)
block = Semaphore(0)
must_wait = False
```

`eating` and `waiting` keep track of the number of threads sitting at the bar and waiting. `mutex` protects both counters. `must_wait` indicates that the bar is (or has been) full, som incoming customers have to block on `block`.

7.1.2 Sushi bar non-solution

Here is an incorrect solution Reek uses to illustrate one of the difficulties of this problem.

Sushi bar non-solution

```
mutex.wait()
if must_wait:
    waiting += 1
    mutex.signal()
    block.wait()

    mutex.wait()      # reacquire mutex
    waiting -= 1

eating += 1
must_wait = (eating == 5)
mutex.signal()

# eat sushi

mutex.wait()
eating -= 1
if eating == 0:
    n = min(5, waiting)
    block.signal(n)
    must_wait = False
mutex.signal()
```

Puzzle: what's wrong with this solution?

7.1.3 Sushi bar non-solution

The problem is at Line 7. If a customer arrives while the bar is full, he has to give up the mutex while he waits so that other customers can leave. When the last customer leaves, she signals `block`, which wakes up at least some of the waiting customers, and clears `must_wait`.

But when the customers wake up, they have to get the mutex back, and that means they have to compete with incoming new threads. If new threads arrive and get the mutex first, they could take all the seats before the waiting threads. This is not just a question of injustice; it is possible for more than 5 threads to be in the critical section concurrently, which violates the synchronization constraints.

Reek provides two solutions to this problem, which appear in the next two sections.

Puzzle: see if you can come up with two different correct solutions!

Hint: neither solution uses any additional variables.

7.1.4 Sushi bar solution #1

The only reason a waiting customer has to reacquire the mutex is to update the state of `eating` and `waiting`, so one way to solve the problem is to make the departing customer, who already has the mutex, do the updating.

Sushi bar solution #1

```
mutex.wait()
if must_wait:
    waiting += 1
    mutex.signal()
    block.wait()
else:
    eating += 1
    must_wait = (eating == 5)
    mutex.signal()

# eat sushi

mutex.wait()
eating -= 1
if eating == 0:
    n = min(5, waiting)
    waiting -= n
    eating += n
    must_wait = (eating == 5)
    block.signal(n)
mutex.signal()
```

When the last departing customer releases the mutex, `eating` has already been updated, so newly arriving customers see the right state and block if necessary. Reek calls this pattern "I'll do it for you," because the departing thread is doing work that seems, logically, to belong to the waiting threads.

A drawback of this approach is that is it a little more difficult to confirm that the state is being updated correctly.

7.1.5 Sushi bar solution #2

Reek's alternative solution is based on the counterintuitive notion that we can transfer a mutex from one thread to another! In other words, one thread can acquire a lock and then another thread can release it. As long as both threads understand that the lock has been transferred, there is nothing wrong with this.

Sushi bar solution #2

```
mutex.wait()
if must_wait:
    waiting += 1
    mutex.signal()
    block.wait()      # when we resume, we have the mutex
    waiting -= 1

eating += 1
must_wait = (eating == 5)
if waiting and not must_wait:
    block.signal()            # and pass the mutex
else:
    mutex.signal()

# eat sushi

mutex.wait()
eating -= 1
if eating == 0: must_wait = False

if waiting and not must_wait:
    block.signal()            # and pass the mutex
else:
    mutex.signal()
```

If there are fewer than 5 customers at the bar and no one waiting, an entering customer just increments `eating` and releases the mutex. The fifth customer sets `must_wait`.

If `must_wait` is set, entering customers block until the last customer at the bar clears `must_wait` and signals `block`. It is understood that the signaling thread gives up the mutex and the waiting thread receives it. Keep in mind, though, that this is an invariant understood by the programmer, and documented in the comments, but not enforced by the semantics of semaphores. It is up to us to get it right.

When the waiting thread resumes, we understand that it has the mutex. If there are other threads waiting, it signals `block` which, again, passes the mutex to a waiting thread. This process continues, with each thread passing the mutex to the next until there are no more chairs or no more waiting threads. In either case, the last thread releases the mutex and goes to sit down.

Reek calls this pattern "Pass the baton," since the mutex is being passed from one thread to the next like a baton in a relay race. One nice thing about this solution is that it is easy to confirm that updates to `eating` and `waiting` are consistent. A drawback is that it is harder to confirm that the mutex is being used correctly.

7.2 The child care problem

Max Hailperin wrote this problem for his textbook *Operating Systems and Middleware* [5]. At a child care center, state regulations require that there is always one adult present for every three children.

Puzzle: Write code for child threads and adult threads that enforces this constraint in a critical section.

7.2.1 Child care hint

Hailperin suggests that you can *almost* solve this problem with one semaphore.

Child care hint

```
multiplex = Semaphore(0)
```

`multiplex` counts the number of tokens available, where each token allows a child thread to enter. As adults enter, they signal `multiplex` three times; as they leave, they `wait` three times. But there is a problem with this solution.

Puzzle: what is the problem?

7.2.2 Child care non-solution

Here is what the adult code looks like in Hailperin's non-solution:

Child care non-solution (adult)

```
multiplex.signal(3)

# critical section

multiplex.wait()
multiplex.wait()
multiplex.wait()
```

The problem is a potential deadlock. Imagine that there are three children and two adults in the child care center. The value of `multiplex` is 3, so either adult should be able to leave. But if both adults start to leave at the same time, they might divide the available tokens between them, and both block.

Puzzle: solve this problem with a minimal change.

7.2.3 Child care solution

Adding a mutex solves the problem:

Child care solution (adult)

```
multiplex.signal(3)

# critical section

mutex.wait()
    multiplex.wait()
    multiplex.wait()
    multiplex.wait()
mutex.signal()
```

Now the three `wait` operations are atomic. If there are three tokens available, the thread that gets the mutex will get all three tokens and exit. If there are fewer tokens available, the first thread will block in the mutex and subsequent threads will queue on the mutex.

7.2.4 Extended child care problem

One feature of this solution is that an adult thread waiting to leave can prevent child threads from entering.

Imagine that there are 4 children and two adults, so the value of the multiplex is 2. If one of the adults tries to leave, she will take two tokens and then block waiting for the third. If a child thread arrives, it will wait even though it would be legal to enter. From the point of view of the adult trying to leave, that might be just fine, but if you are trying to maximize the utilization of the child care center, it's not.

Puzzle: write a solution to this problem that avoids unnecessary waiting.

Hint: think about the dancers in Section 3.8.

7.2.5 Extended child care hint

Here are the variables I used in my solution:

Extended child care hint

```
children = adults = waiting = leaving = 0
mutex = Semaphore(1)
childQueue = Semaphore(0)
adultQueue = Semaphore(0)
```

`children`, `adults`, `waiting` and `leaving` keep track of the number of children, adults, children waiting to enter, and adults waiting to leave; they are protected by `mutex`.

Children wait on `childQueue` to enter, if necessary. Adults wait on `adultQueue` to leave.

7.2.6 Extended child care solution

This solution is more complicated than Hailperin's elegant solution, but it is mostly a combination of patterns we have seen before: a scoreboard, two queues, and "I'll do it for you".

Here is the child code:

Extended child care solution (child)

```
mutex.wait()
    if children < 3 * adults:
        children++
        mutex.signal()
    else:
        waiting++
        mutex.signal()
        childQueue.wait()

# critical section

mutex.wait()
    children--
    if leaving and children <= 3 * (adults-1):
        leaving--
        adults--
        adultQueue.signal()
mutex.signal()
```

As children enter, they check whether there are enough adults and either (1) increment `children` and enter or (2) increment `waiting` and block. When they exit, they check for an adult thread waiting to leave and signal it if possible.

Here is the code for adults:

Extended child care solution (adult)

```
mutex.wait()
    adults++
    if waiting:
        n = min(3, waiting)
        childQueue.signal(n)
        waiting -= n
        children += n
mutex.signal()

# critical section

mutex.wait()
    if children <= 3 * (adults-1):
        adults--
        mutex.signal()
    else:
        leaving++
        mutex.signal()
        adultQueue.wait()
```

As adults enter, they signal waiting children, if any. Before they leave, they check whether there are enough adults left. If so, they decrement `adults` and exit. Otherwise they increment `leaving` and block. While an adult thread is waiting to leave, it counts as one of the adults in the critical section, so additional children can enter.

7.3 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 out 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.

7.3.1 Room party hint

Room party hint

```
students = 0
dean = 'not here'
mutex = Semaphore(1)
turn = Semaphore(1)
clear = Semaphore(0)
lieIn = Semaphore(0)
```

`students` counts the number of students in the room, and `dean` is the state of the Dean, which can also be "waiting" or "in the room". `mutex` protects `students` and `dean`, so this is yet another example of a scoreboard.

`turn` is a turnstile that keeps students from entering while the Dean is in the room.

`clear` and `lieIn` are used as rendezvouses between a student and the Dean (which is a whole other kind of scandal!).

7.3.2 Room party solution

This problem is hard. I worked through a lot of versions before I got to this one. The version that appeared in the first edition was mostly correct, but occasionally the Dean would enter the room and then find that he could neither search nor break up the party, so he would have to skulk off in embarrassed silence.

Matt Tesch wrote a solution that spared this humiliation, but the result was complicated enough that we had a hard time convincing ourselves that it was correct. But that solution led me to this one, which is a bit more readable.

Room party solution (dean)

```
mutex.wait()
    if students > 0 and students < 50:
        dean = 'waiting'
        mutex.signal()
        lieIn.wait()     # and get mutex from the student.

    # students must be 0 or >= 50

    if students >= 50:
        dean = 'in the room'
        breakup()
        turn.wait()      # lock the turnstile
        mutex.signal()
        clear.wait()     # and get mutex from the student.
        turn.signal()    # unlock the turnstile

    else:                # students must be 0
        search()

dean = 'not here'
mutex.signal()
```

When the Dean arrives, there are three cases: if there are students in the room, but not 50 or more, the Dean has to wait. If there are 50 or more, the Dean breaks up the party and waits for the students to leave. If there are no students, the Dean searches and leaves.

In the first two cases, the Dean has to wait for a rendezvous with a student, so he has to give up `mutex` to avoid a deadlock. When the Dean wakes up, he has to modify the scoreboard, so he needs to get the mutex back. This is similar to the situation we saw in the Sushi Bar problem. The solution I chose is the "Pass the baton" pattern.

Room party solution (student)

```
1  mutex.wait()
2      if dean == 'in the room':
3          mutex.signal()
4          turn.wait()
5          turn.signal()
6          mutex.wait()
7
8      students += 1
9
10     if students == 50 and dean == 'waiting':
11         lieIn.signal()
   # and pass mutex to the dean
12     else:
13         mutex.signal()
14
15 party()
16
17 mutex.wait()
18     students -= 1
19
20     if students == 0 and dean == 'waiting':
21         lieIn.signal()          # and pass mutex to the dean
22     elif students == 0 and dean == 'in the room':
23         clear.signal()          # and pass mutex to the dean
24     else:
25         mutex.signal()
```

There are three cases where a student might have to signal the Dean. If the Dean is waiting, then the 50th student in or the last one out has to signal `lieIn`. If the Dean is in the room (waiting for all the students to leave), the last student out signals `clear`. In all three cases, it is understood that the mutex passes from the student to the Dean.

One part of this solution that may not be obvious is how we know at Line 7 of the Dean's code that `students` must be 0 or not less than 50. The key is to realize that there are only two ways to get to this point: either the first conditional was false, which means that `students` is either 0 or not less than 50; or the Dean was waiting on `lieIn` when a student signaled, which means, again, that `students` is either 0 or not less than 50.

7.4 The Senate 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.

7.4.1 Bus problem hint

Here are the variables I used in my solution:

Bus problem hint

```
riders = 0
mutex = Semaphore(1)
multiplex = Semaphore(50)
bus = Semaphore(0)
allAboard = Semaphore(0)
```

`mutex` protects `riders`, which keeps track of how many riders are waiting; `multiplex` makes sure there are no more than 50 riders in the boarding area.

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

7.4.2 Bus problem solution #1

Here is the code for the bus. Again, we are using the "Pass the baton" pattern.

Bus problem solution (bus)

```
mutex.wait()
if riders > 0:
    bus.signal()         # and pass the mutex
    allAboard.wait()     # and get the mutex back
mutex.signal()

depart()
```

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

Here is the code for the riders:

Bus problem solution (riders)

```
multiplex.wait()
    mutex.wait()
        riders += 1
    mutex.signal()

    bus.wait()               # and get the mutex
multiplex.signal()

boardBus()

riders -= 1
if riders == 0:
    allAboard.signal()
else:
    bus.signal()             # and pass the mutex
```

The multiplex controls the number of riders in the waiting area, although strictly speaking, a rider doesn't enter the waiting area until she increments `riders`.

Riders wait on `bus` until the bus arrives. When a rider wakes up, it is understood that she has the mutex. After boarding, each rider decrements `riders`. If there are more riders waiting, the boarding rider signals `bus` and pass the mutex to the next rider. The last rider signals `allAboard` and passes the mutex back to the bus.

Finally, the bus releases the mutex and departs.

Puzzle: can you find a solution to this problem using the "I'll do it for you" pattern?

7.4.3 Bus problem solution #2

Grant Hutchins came up with this solution, which uses fewer variables than the previous one, and doesn't involve passing around any mutexes. Here are the variables:

Bus problem solution #2 (initialization)

```
waiting = 0
mutex = new Semaphore(1)
bus = new Semaphore(0)
boarded = new Semaphore(0)
```

`waiting` is the number of riders in the boarding area, which is protected by `mutex`. `bus` signals when the bus has arrived; `boarded` signals that a rider has boarded.

Here is the code for the bus.

Bus problem solution (bus)

```
mutex.wait()
n = min(waiting, 50)
for i in range(n):
    bus.signal()
    boarded.wait()

waiting = max(waiting-50, 0)
mutex.signal()

depart()
```

The bus gets the `mutex` and holds it throughout the boarding process. The loop signals each rider in turn and waits for her to board. By controlling the number of signals, the bus prevents more than 50 riders from boarding.

When all the riders have boarded, the bus updates `waiting`, which is an example of the "I'll do it for you" pattern.

The code for the riders uses two simple patterns: a mutex and a rendezvous.

Bus problem solution (riders)

```
mutex.wait()
     waiting += 1
mutex.signal()

bus.wait()
board()
boarded.signal()
```

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?

7.5 The Faneuil Hall problem

This problem was written by Grant Hutchins, who was inspired by a friend who took her Oath of Citizenship at Faneuil Hall in Boston.

"There are three kinds of threads: immigrants, spectators, and a one judge. Immigrants must wait in line, check in, and then sit down. At some point, the judge enters the building. When the judge is in the building, no one may enter, and the immigrants may not leave. Spectators may leave. Once all immigrants check in, the judge can confirm the naturalization. After the confirmation, the immigrants pick up their certificates of U.S. Citizenship. The judge leaves at some point after the confirmation. Spectators may now enter as before. After immigrants get their certificates, they may leave."

To make these requirements more specific, let's give the threads some functions to execute, and put constraints on those functions.

- Immigrants must invoke `enter`, `checkIn`, `sitDown`, `swear`, `getCertificate` and `leave`.
- The judge invokes `enter`, `confirm` and `leave`.
- Spectators invoke `enter`, `spectate` and `leave`.
- While the judge is in the building, no one may `enter` and immigrants may not `leave`.
- The judge can not `confirm` until all immigrants who have invoked `enter` have also invoked `checkIn`.
- Immigrants can not `getCertificate` until the judge has executed `confirm`.

7.5.1 Faneuil Hall Problem Hint

Faneuil Hall problem hint

```
noJudge = Semaphore(1)
entered = 0
checked = 0
mutex = Semaphore(1)
confirmed = Semaphore(0)
```

`noJudge` acts as a turnstile for incoming immigrants and spectators; it also protects `entered`, which counts the number of immigrants in the room. `checked` counts the number of immigrants who have checked in; it is protected by `mutex`.

`confirmed` signals that the judge has executed `confirm`.

7.5.2 Faneuil Hall problem solution

Here is the code for immigrants:

Faneuil Hall problem solution (immigrant)

```
noJudge.wait()
enter()
entered++
noJudge.signal()

mutex.wait()
checkIn()
checked++

if judge = 1 and entered == checked:
    allSignedIn.signal()
# and pass the mutex
else:
    mutex.signal()

sitDown()
confirmed.wait()

swear()
getCertificate()

noJudge.wait()
leave()
noJudge.signal()
```

Immigrants pass through a turnstile when they enter; while the judge is in the room, the turnstile is locked.

After entering, immigrants have to get `mutex` to check in and update `checked`. If there is a judge waiting, the last immigrant to check in signals `allSignedIn` and passes the mutex to the judge.

Here is the code for the judge:

Faneuil Hall problem solution (judge)

```
noJudge.wait()
mutex.wait()

enter()
judge = 1

if entered > checked:
    mutex.signal()
```

```
    allSignedIn.wait()
# and get the mutex back.

confirm()
confirmed.signal(checked)
entered = checked = 0

leave()
judge = 0

mutex.signal()
noJudge.signal()
```

The judge holds `noJudge` to bar immigrants and spectators from entering, and `mutex` so he can access `entered` and `checked`.

If the judge arrives at an instant when everyone who has entered has also checked in, she can proceed immediately. Otherwise, she has to give up the mutex and wait. When the last immigrant checks in and signals `allSignedIn`, it is understood that the judge will get the mutex back.

After invoking `confirm`, the judge signals `confirmed` once for every immigrant who has checked in, and then resets the counters (an example of "I'll do it for you"). Then the judge leaves and releases `mutex` and `noJudge`.

After the judge signals `confirmed`, immigrants invoke `swear` and `getCertificate` concurrently, and then wait for the `noJudge` turnstile to open before leaving.

The code for spectators is easy; the only constraint they have to obey is the `noJudge` turnstile.

Faneuil Hall problem solution (spectator)

```
noJudge.wait()
enter()
noJudge.signal()

spectate()

leave()
```

Note: in this solution it is possible for immigrants to get stuck, after they get their certificate, by another judge coming to swear in the next batch of immigrants. If that happens, they might have to wait through another swearing in-ceremony.

Puzzle: modify this solution to handle the additional constraint that after the judge leaves, all immigrants who have been sworn in must leave before the judge can enter again.

7.5.3 Extended Faneuil Hall Problem Hint

My solution uses the following additional variables:

Faneuil Hall problem hint

```
exit = Semaphore(0)
allGone = Semaphore(0)
```

Since the extended problem involves an additional rendezvous, we can solve it with two semaphores.

One other hint: I found it useful to use the "pass the baton" pattern again.

7.5.4 Extended Faneuil Hall problem solution

The top half of this solution is the same as before. The difference starts at Line 21. Immigrants wait here for the judge to leave.

Faneuil Hall problem solution (immigrant)

```
1  noJudge.wait()
2  enter()
3  entered++
4  noJudge.signal()
5
6  mutex.wait()
7  checkIn()
8  checked++
9
10 if judge = 1 and entered == checked:
11     allSignedIn.signal()
   # and pass the mutex
12 else:
13     mutex.signal()
14
15 sitDown()
16 confirmed.wait()
17
18 swear()
19 getCertificate()
20
21 exit.wait()
   # and get the mutex
22 leave()
23 checked--
24 if checked == 0:
25     allGone.signal()
   # and pass the mutex
26 else:
27     exit.signal()
   # and pass the mutex
```

For the judge, the difference starts at Line 18. When the judge is ready to leave, she can't release `noJudge`, because that would allow more immigrants, and possibly another judge, to enter. Instead, she signals `exit`, which allows one immigrant to leave, and passes `mutex`.

The immigrant that gets the signal decrements `checked` and then passes the baton to the next immigrant. The last immigrant to leave signals `allGone` and passes the mutex back to the judge. This pass-back is not strictly necessary, but it has the nice feature that the judge releases both `mutex` and `noJudge` to end the phase cleanly.

Faneuil Hall problem solution (judge)

```
noJudge.wait()
mutex.wait()

enter()
judge = 1

if entered > checked:
    mutex.signal()
    allSignedIn.wait()
# and get the mutex back.

confirm()
confirmed.signal(checked)
entered = 0

leave()
judge = 0

exit.signal()
# and pass the mutex
allGone.wait()                      # and get it back
mutex.signal()
noJudge.signal()
```

The spectator code for the extended problem is unchanged.

7.6 Dining Hall problem

This problem was written by Jon Pollack during my Synchronization class at Olin College.

Students in the dining hall invoke `dine` and then `leave`. After invoking `dine` and before invoking `leave` a student is considered "ready to leave".

The synchronization constraint that applies to students is that, in order to maintain the illusion of social suave, a student may never sit at a table alone. A student is considered to be sitting alone if everyone else who has invoked `dine` invokes `leave` before she has finished `dine`.

Puzzle: write code that enforces this constraint.

7.6.1 Dining Hall problem hint

Dining Hall problem hint

```
eating = 0
readyToLeave = 0
mutex = Semaphore(1)
okToLeave = Semaphore(0)
```

`eating` and `readyToLeave` are counters protected by `mutex`, so this is the usual scoreboard pattern.

If a student is ready to leave, but another student would be left alone at the table, she waits on `okToLeave` until another student changes the situation and signals.

7.6.2 Dining Hall problem solution

If you analyze the constraints, you will realize that there is only one situation where a student has to wait, if there is one student eating and one student who wants to leave. But there are two ways to get out of this situation: another student might arrive to eat, or the dining student might finish.

In either case, the student who signals the waiting student updates the counters, so the waiting student doesn't have to get the mutex back. This is another example of the the "I'll do it for you" pattern.

Dining Hall problem solution

```
getFood()

mutex.wait()
eating++
if eating == 2 and readyToLeave == 1:
    okToLeave.signal()
    readyToLeave--
mutex.signal()

dine()

mutex.wait()
eating--
readyToLeave++

if eating == 1 and readyToLeave == 1:
    mutex.signal()
    okToLeave.wait()
elif eating == 0 and readyToLeave == 2:
    okToLeave.signal()
    readyToLeave -= 2
    mutex.signal()
else:
    readyToLeave--
    mutex.signal()

leave()
```

When is student is checking in, if she sees one student eating and one waiting to leave, she lets the waiter off the hook and decrements `readyToLeave` for him.

After dining, the student checks three cases:

- If there is only one student left eating, the departing student has to give up the mutex and wait.
- If the departing student finds that someone is waiting for her, she signals him and updates the counter for both of them.
- Otherwise, she just decrements `readyToLeave` and leaves.

7.6.3 Extended Dining Hall problem

The Dining Hall problem gets a little more challenging if we add another step. As students come to lunch they invoke `getFood`, `dine` and then `leave`. After invoking `getFood` and before invoking `dine`, a student is considered "ready to eat". Similarly, after invoking `dine` a student is considered "ready to leave".

The same synchronization constraint applies: a student may never sit at a table alone. A student is considered to be sitting alone if either

- She invokes `dine` while there is no one else at the table and no one ready to eat, or
- everyone else who has invoked `dine` invokes `leave` before she has finished `dine`.

Puzzle: write code that enforces these constraints.

7.6.4 Extended Dining Hall problem hint

Here are the variables I used in my solution:

Extended Dining Hall problem hint

```
readyToEat = 0
eating = 0
readyToLeave = 0
mutex = Semaphore(1)
okToSit = Semaphore(0)
okToLeave = Semaphore(0)
```

`readyToEat`, `eating` and `readyToLeave` are counters, all protected by `mutex`.

If a student is in a situation where she cannot proceed, she waits on `okToSit` or `okToLeave` until another student changes the situation and signals.

I also used a per-thread variable named `hasMutex` to help keep track of whether or not a thread holds the mutex.

7.6.5 Extended Dining Hall problem solution

Again, if we analyze the constraints, we realize that there is only one situation where a student who is ready to eat has to wait, if there is no one eating and no one else ready to eat. And the only way out is if someone else arrives who is ready to eat.

Extended Dining Hall problem solution

```
getFood()

mutex.wait()
readyToEat++
if eating == 0 and readyToEat == 1:
    mutex.signal()
    okToSit.wait()
elif eating == 0 and readyToEat == 2:
    okToSit.signal()
    readyToEat -= 2
    eating += 2
    mutex.signal()
else:
    readyToEat--
    eating++
    if eating == 2 and readyToLeave == 1:
        okToLeave.signal()
        readyToLeave--
    mutex.signal()

dine()

mutex.wait()
eating--
readyToLeave++
if eating == 1 and readyToLeave == 1:
    mutex.signal()
    okToLeave.wait()
elif eating == 0 and readyToLeave == 2:
    okToLeave.signal()
    readyToLeave -= 2
    mutex.signal()
else:
    readyToLeave--
    mutex.signal()

leave()
```

As in the previous solution, I used the "I'll do it for you" pattern so that a waiting student doesn't have to get the mutex back.

The primary difference between this solution and the previous one is that the first student who arrives at an empty table has to wait, and the second student allows both students to proceed. It either case, we don't have to check for students waiting to leave, since no one can leave an empty table!