Cigarette smokers problem

The cigarette smokers problem problem was originally presented by Suhas Patil , 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.

To explain the premise, the agent represents an operating system that allocates resources, and the smokers represent applications that need resources. The problem is to make sure that if resources are available that would allow one more more applications to proceed, those applications should be woken up. Conversely, we want to avoid waking an application if it cannot proceed.

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

The impossible version: Patil's version imposes restrictions on the solution. First, you are not allowed to modify the agent code. If the agent represents an operating system, it makes sense to assume that you don't want to modify it every time a new application comes along. The second restriction is that you can't use conditional statements or an array of semaphores. With these constraints, the problem cannot be solved, but as Parnas points out, the second restriction is pretty artificial . With constraints like these, a lot of problems become unsolvable.

The interesting version: This version keeps the first restriction—you can't change the agent code—but it drops the others.

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:

Agent semaphores

```
1 agentSem = Semaphore(1)
2 tobacco = Semaphore(0)
3 paper = Semaphore(0)
4 match = Semaphore(0)
```

The agent is actually made up of three concurrent threads, Agent A, Agent B and Agent C. Each waits on `agentSem`; each time `agentSem` is signaled, one of the Agents wakes up and provides ingredients by signaling two semaphores.

Agent A code

```
1 agentSem.wait()
2 tobacco.signal()
3 paper.signal()
```

Agent B code

```
1 agentSem.wait()
2 paper.signal()
3 match.signal()
```

Agent C code

```
1 agentSem.wait()
2 tobacco.signal()
3 match.signal()
```

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

Smoker with matches

```
1 tobacco.wait()
2 paper.wait()
3 agentSem.signal()
```

Smoker with tobacco

```
1 paper.wait()
2 match.wait()
3 agentSem.signal()
```

Smoker with paper

```
1  tobacco.wait()
2  match.wait()
3  agentSem.signal()
```

What's wrong with this solution?

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!

Smokers 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

Smokers problem hint

```
1  isTobacco = isPaper = isMatch = False
2  tobaccoSem = Semaphore(0)
3  paperSem = Semaphore(0)
4  matchSem = Semaphore(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.

Smoker problem solution

Here is the code for one of the pushers:

Pusher A

```
tobacco.wait()
mutex.wait()
    if isPaper:
        isPaper = False
        matchSem.signal()
    elseif 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:

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.

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.

Generalized Smokers Problem Hint

If the agents don't wait for the smokers, ingredients might accumulate on the table. Instead of using boolean values to keep track of ingredients, we need integers to count them.

Generalized Smokers problem hint

```
1  numTobacco = numPaper = numMatch = 1
```

Generalized Smokers Problem Solution

Here is the modified code for Pusher A:

Pusher A

```
1  tobacco.wait()
2  mutex.wait()
3      if numPaper:
4          numPaper -= 1
5          matchSem.signal()
6      elseif numMatch:
7          numMatch -= 1
8          paperSem.signal()
9      else:
10         numTobacco += 1
11 mutex.signal()
```

One way to visualize this problem is to imagine that when an Agent runs, it creates two pushers, gives each of them one ingredient, and puts them in a room with all the other pushers. Because of the mutex, the pushers file into a room where there are three sleeping smokers and a table. One at a time, each pusher enters the room and checks the ingredients on the table. If he can assemble a complete set of ingredients, he takes them off the table and wakes the corresponding smoker. If not, he leaves his ingredient on the table and leaves without waking anyone.

This is an example of a pattern we will see several times, which I call a **scoreboard**. The variables `numPaper`, `numTobacco` and `numMatch` keep track of the state of the system. As each thread files through the mutex, it checks the state as if looking at the scoreboard, and reacts accordingly.

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.
- Barber threads should invoke `cutHair`.
- When the barber invokes `cutHair` there should be exactly one thread invoking `getHairCut` concurrently.

Barbershop hint

Barbershop hint

```
1  customers = 0
2  mutex = Semaphore(1)
3  customer = Semaphore(0)
4  barber = Semaphore(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.

Barbershop solution

Again, my solution combines a scoreboard and a rendezvous. Here is the code for customers:

Barbershop solution (customer)

```
1  mutex.wait()
2      if customers == n+1:
3          mutex.signal()
4          balk()
5      customers += 1
6  mutex.signal()
7
8  customer.signal()
9  barber.wait()
10 getHairCut()
11
12 mutex.wait()
13     customers -= 1
14 mutex.signal()
```

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 `balk`. Otherwise each customer signals `customer` and waits on `barber`.

Here is the code for barbers:

Barbershop solution (barber)

```
1  customer.wait()
2  barber.signal()
3  cutHair()
```

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."

Hilzer's Barbershop problem

William Stallings [ ] 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`, `sitInBarberChair`, `pay`, `exitShop`.
- 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` 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 as defined in Section 3.8.

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

Hilzer's barbershop hint

Here are the variables I used in my solution:

Hilzer's barbershop hint

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

Hilzer's barbershop solution

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

Hilzer's barbershop solution (customer)

```
mutex.wait()
    if customers == 20:
        mutex.signal()
        exitShop()
    customers += 1
mutex.signal()

standingRoom.wait()
enterShop()

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

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

```

```
19  customer.signal()
20  barber.wait()
21  getHairCut()
22
23  pay()
24  cash.signal()
25  receipt.wait()
26
27  mutex.wait()
28      customers -= 1
29  mutex.signal()
30
31  exitShop()
```

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`; as each thread moves to the next stage, it signals the previous stage.

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

The code for the barber is self-explanatory:

Hilzer's barbershop solution (barber)

```
1  customer.wait()
2  barber.signal()
3  cutHair()
4
5  cash.wait()
6  acceptPayment()
7  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.

The Santa Claus problem

This problem is from William Stallings's *Operating Systems* , 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.

Santa problem hint

Santa problem hint

```
1  elves = 0
2  reindeer = 0
3  santaSem = Semaphore(0)
4  reindeerSem = Semaphore(0)
5  elfTex = Semaphore(1)
6  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.

Santa problem solution

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

Santa problem solution (Santa)

```
1  santaSem.wait()
2  mutex.wait()
3      if reindeer == 9:
4          prepareSleigh()
5          reindeerSem.signal(9)
6      else if elves == 3:
7          helpElves()
8  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:

Santa problem solution (reindeer)

```
1  mutex.wait()
2      reindeer += 1
3      if reindeer == 9:
4          santaSem.signal()
5  mutex.signal()
6
7  reindeerSem.wait()
8  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.