4.5 Cigarette smokers problem

The cigarette smokers problem problem was originally presented by Suhas Patil [8], 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 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 [7]. 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

```
agentSem = Semaphore(1)
tobacco = Semaphore(0)
paper = Semaphore(0)
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

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

Agent B code

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

Agent C code

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

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

Smoker with matches

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

Smoker with tobacco

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

Smoker with paper

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

What's wrong with this solution?

4.5.1 Deadlock #6

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

4.5.2 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

```
isTobacco = isPaper = isMatch = False
tobaccoSem = Semaphore(0)
paperSem = Semaphore(0)
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.

4.5.3 Smoker problem solution

Here is the code for one of the pushers:

Pusher A

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

4.5.4 Generalized Smokers Problem

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

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

4.5.5 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

```
numTobacco = numPaper = numMatch = 0
```

4.5.6 Generalized Smokers Problem Solution

Here is the modified code for Pusher A:

Pusher A

```
tobacco.wait()
mutex.wait()
    if numPaper:
        numPaper -= 1
        matchSem.signal()
    elif numMatch:
        numMatch -= 1
        paperSem.signal()
    else:
        numTobacco += 1
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.