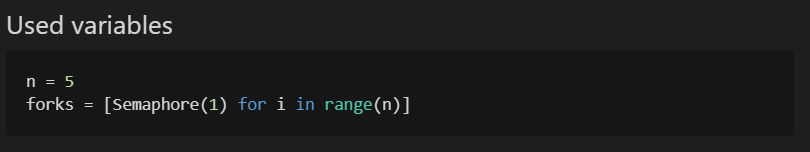
## Week 3

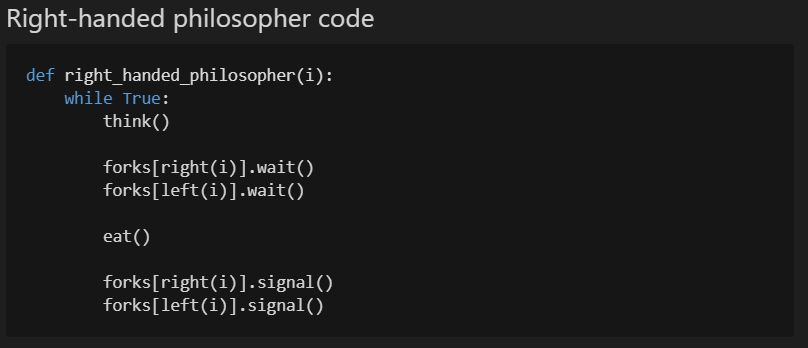
### H 4.4.4 dining philosophers (I)

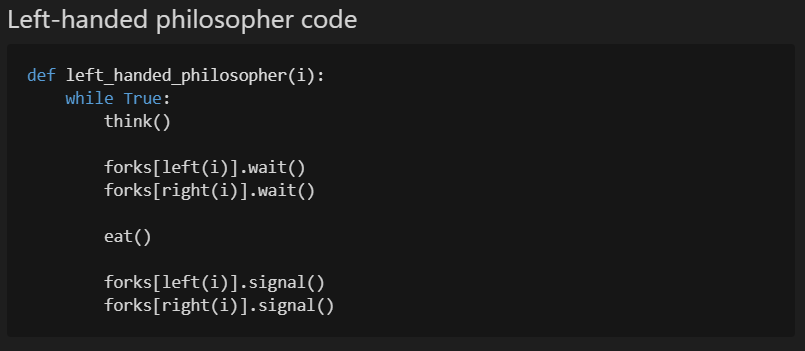
Implement the solution as suggested in 4.4.4 such that there is no circular-waiting (take the standard 4.4 solution as your starting point).

**Solution:**

We solved this problem with an implementation of the dining philosophers code that breaks the circular waiting in the original solution. This is done by having at least 1 “left handed” and “right handed” philosopher at the table. “handedness” in this case means which fork is picked up first. Because of this asymmetry there is no possibility that we end up in a situation where all philosophers are holding a fork and waiting for another.







### I 5.5 Santa Clause

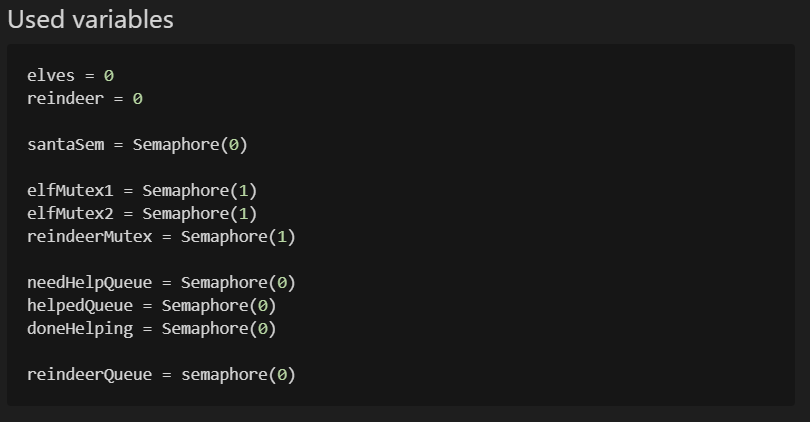
Implement with slightly different requirement: with helping *at least* 3 elves, and reindeer don't have priority. Furthermore: the elf's getHelp() can only be executed together with Santa's helpElves() (so: an explicit synchronization has to be added which is not listed in the book)

Tip: please check first what goes wrong in the LBoS-solution when the reindeer/elf priority has been swapped.

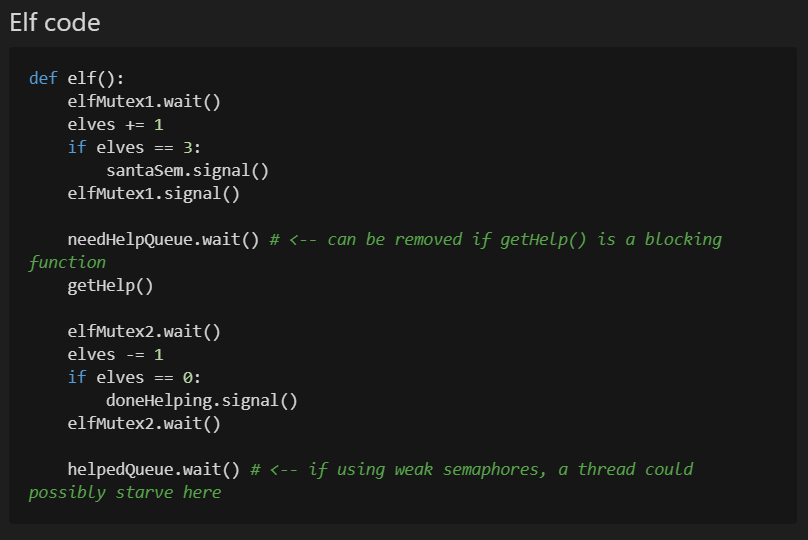
Ensure that an arbitrary number of elf threads can be started (e.g. N=7)

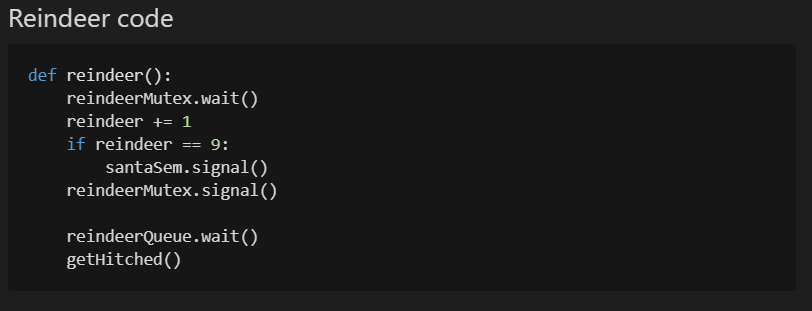
**Solution:**

For this we created 2 different solutions. The first solutions solves the problem of starvation by implementing 2 different mutexes, one for the elves and one for the reindeer. These work independently of one another. The drawback to this solution is that Santa can be woken up twice, in which case the second iteration is redundant, though we consider this a small drawback.

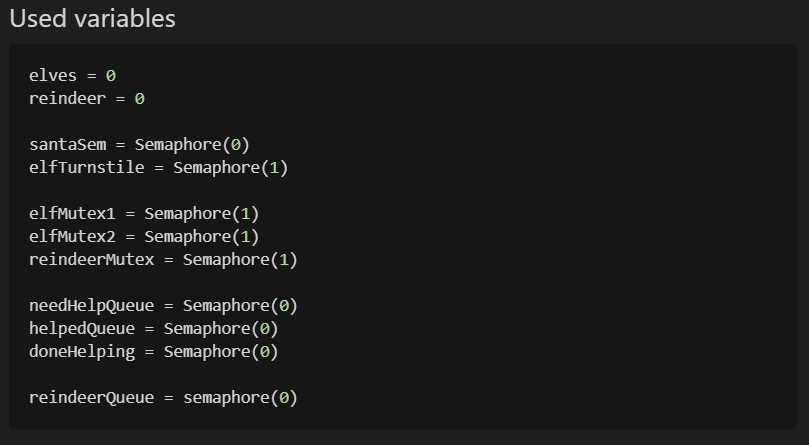


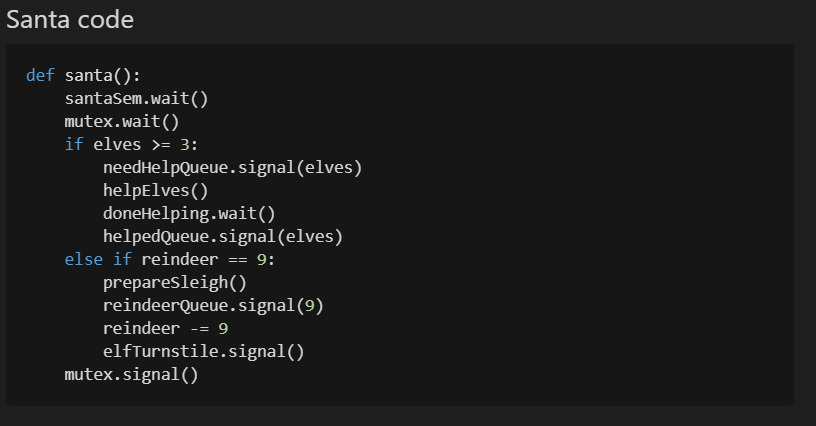




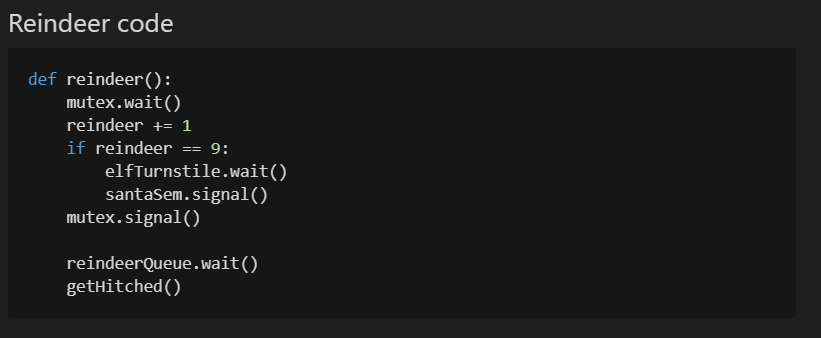


The second solution is more similar to the one from the book, but uses a turnstile for the elves controlled by the reindeer and Santa to prevent the elves from potentially starving the reindeer.









### J 5.6 H2O

Implement without counters (but semaphores, mutexes, pipets, queues, barriers are allowed).

Ensure that an arbitrary number of H and O threads can be started (e.g. N=7).

**Solution:**

We solved this problem by using a barrier with size 3, a semaphore and a mutex/pipet. The mutex/pipet is used to control and limit the flow of the threads. Then we use the semaphore to only allow the oxygen thread to proceed once 2 hydrogen threads are present. Finally, we use the barrier to make sure everything is released at the same time.

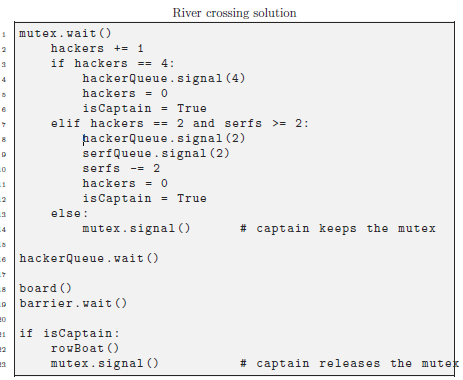


### K 5.7 river crossing

Implement with a symmetric mutex usage (but counters, semaphores, mutexes, pipets, queues, barriers are allowed).

Ensure that an arbitrary number of hacker and serf threads can be started (e.g. N=7).

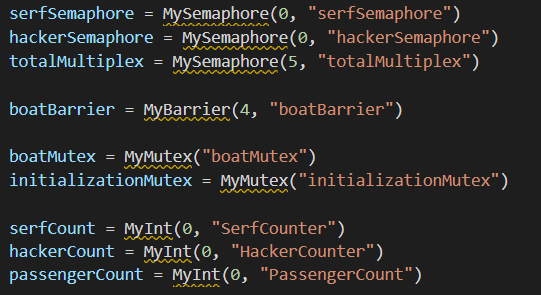
Tip: do not start with the code as given in LBoS and move some statements around until it more or less seems to work, but start with an empty sheet and write a clean implementation.



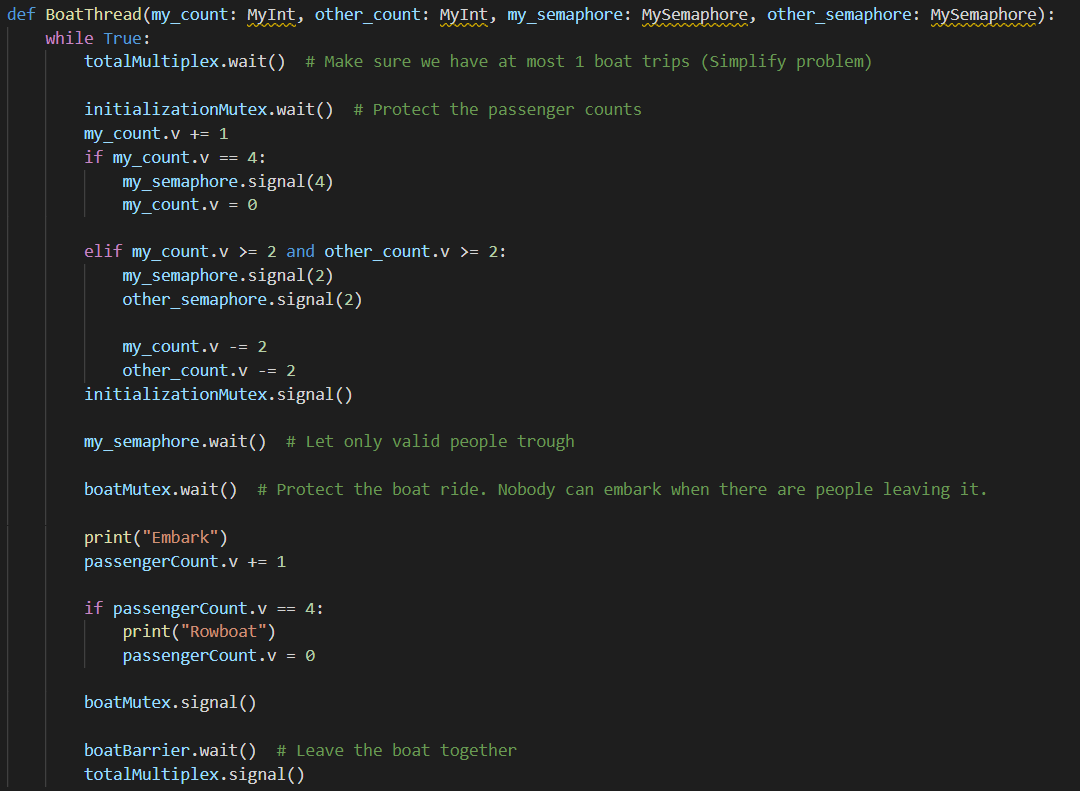
**Solution:**

We were able to solve this problem in a symmetric manner by creating multiple “checkpoints” using mutexes, a turnstile to control the number of passengers who can embark and a multiplex to control the number of passengers who can register for a boat trip.

For this we used the following variables:



And created the following function:



The first thing that this code does is allow only five threads to enter at the same time. This makes sure that only one boatload of threads can pass through. If you would allow four threads. It would block because you have the possibility of having three hackers and one serf or thee serfs and one hacker. This is where the multiplex is used for.

We then count the number of serfs and hackers if we find a valid configuration then we allow those threads through the turnstile. In our code, it is passed as my\_semaphore.

We then count the number of threads that have embarked. The last on rows the boat. Our last step is to have all threads at the barrier. This means all threads have disembarked. We then allow new threads to register as passengers.

Which is called as follows:

