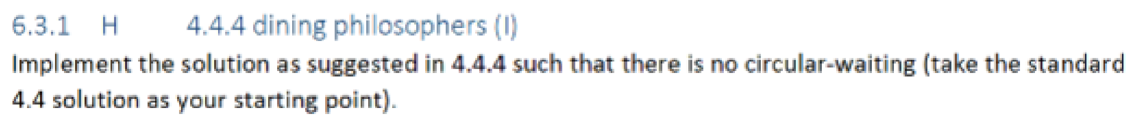
Week 3 - SYNC Assignment

Group 3

Table of contents

First Assignment

Problem description



6.3.1 H 4.4.4 dining philosophers (I)

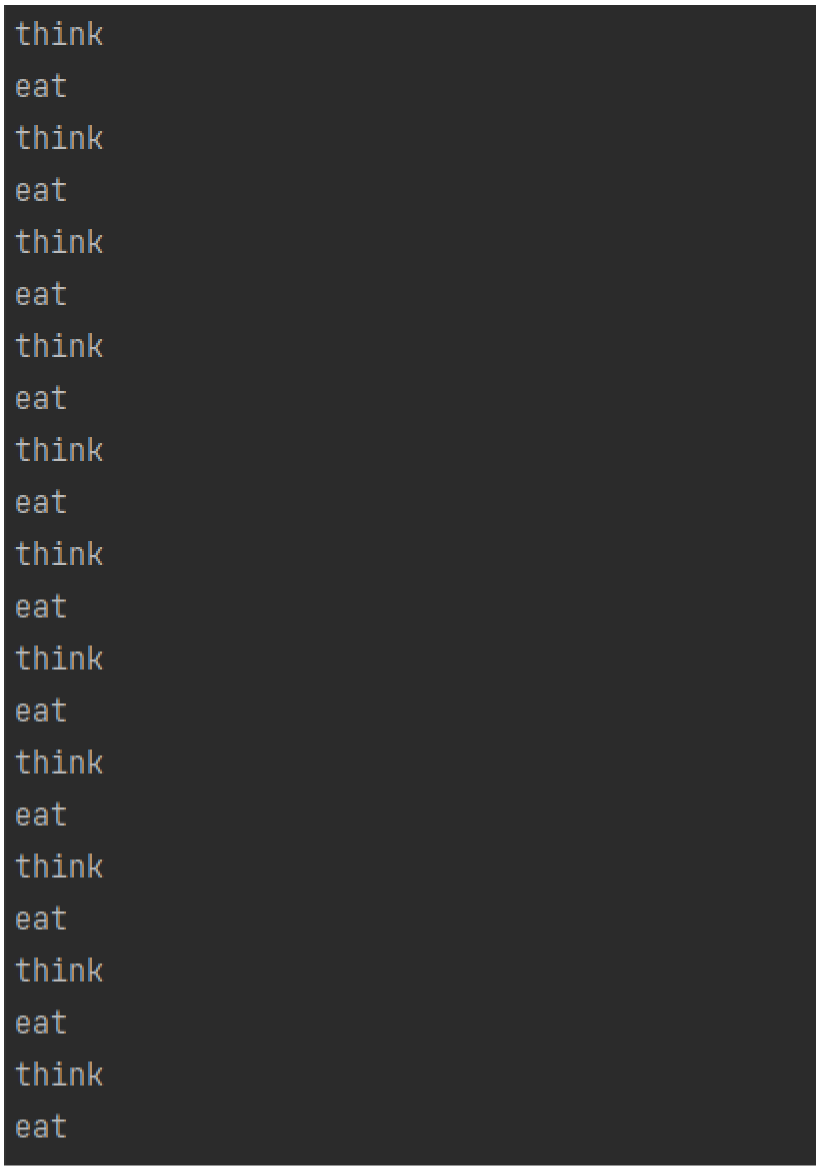
Implement the solution as suggested in 4.4.4 such that there is no circular-waiting (take the standard 4.4 solution as your starting point).

Solution

In this exercise, we solved this problem that breaks the circular waiting in the solution of the book (4.4 section). That is done by having at least one “right-handed” and one “left-handed” philosopher at the table. In this case, “handedness” refers to which fork is picked up first. Because of this asymmetry, we cannot get into a situation where all philosophers are holding a fork and waiting for the other.

```python
forks = [MySemaphore(1, f"Sem{i}") for i in range(5)]

def left(i): return i

def right(i): return (i + 1) % 5

def right_handed(i):
   while True:
       print("think")
       forks[right(i)].wait()
       forks[left(i)].wait()
       print("eat")
       forks[right(i)].signal()
       forks[left(i)].signal()


def left_handed(i):
   while True:
       print("think")
       forks[left(i)].wait()
       forks[right(i)].wait()
       print("eat")
       forks[left(i)].signal()
       forks[right(i)].signal()
```

Output

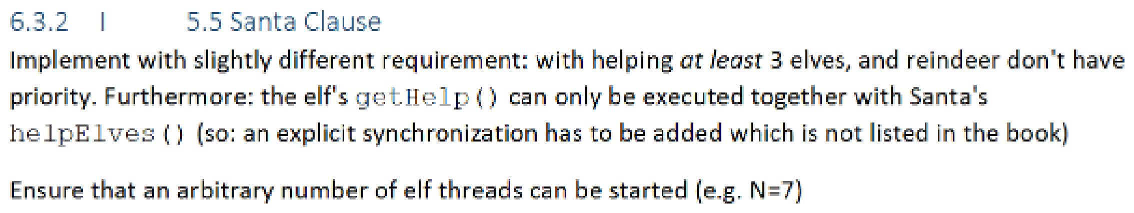


```
think
eat
think
eat
think
eat
think
eat
think
eat
think
eat
think
eat
think
eat
think
eat
think
eat
think
eat
```

A video is also included alongside this document.

Second Assignment

Problem description


6.3.2 | 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)

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

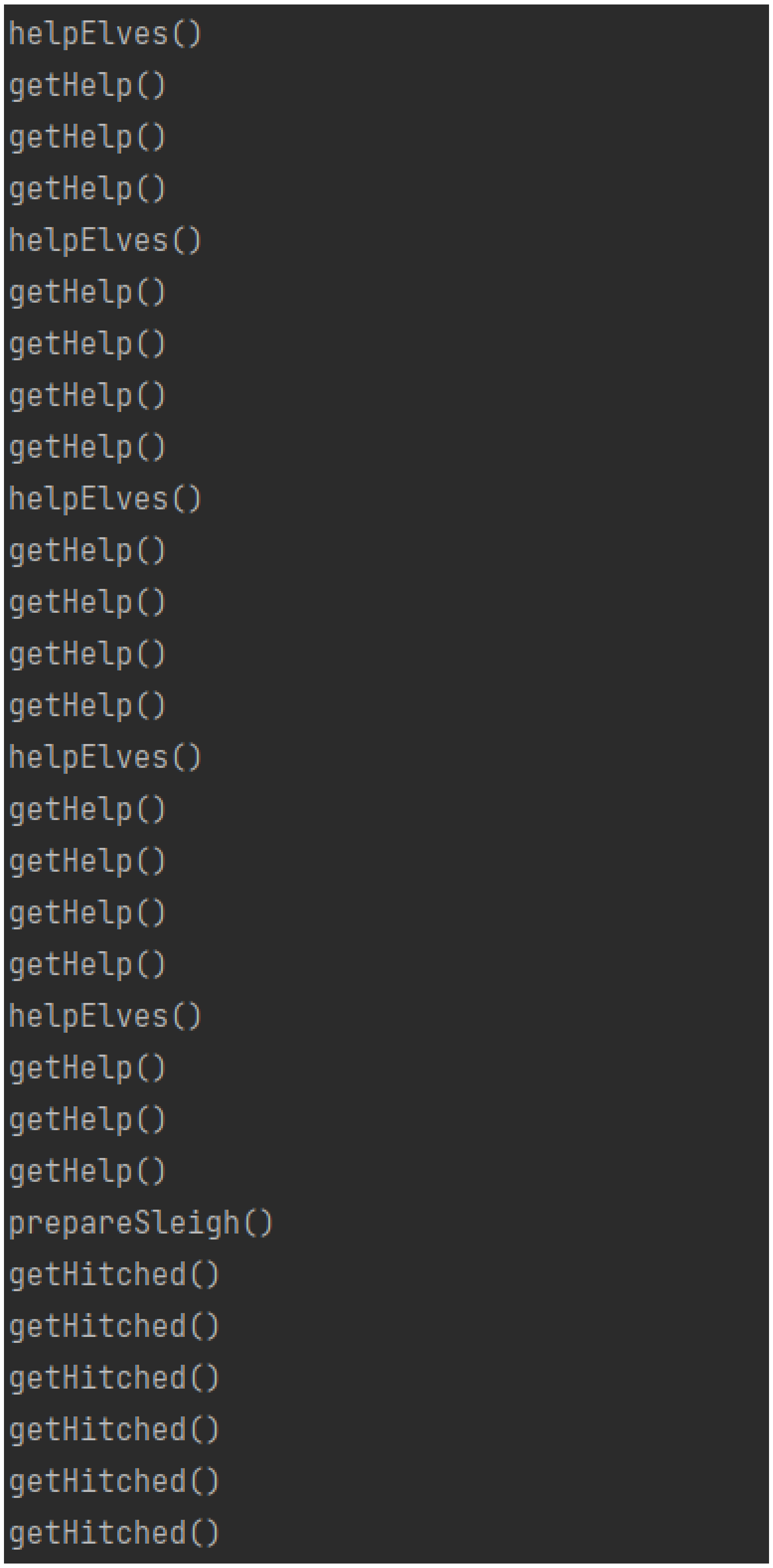
Solution

For this assignment, we had to change the code given in LBoS for the Santa Clause problem such that reindeers do not have priority and that at least 3 elves can get help from Santa. First we create 3 threads (for Santa, for elves and for reindeers). Initially Santa is sleeping, waiting for at least 3 elves so that he can help them. (We assumed that by having priority, it meant that the number of elves should always be checked first). If there are not at least 3 elves, we then check for the reindeers. In order to secure the counters for “elves” and “reindeers”, we used mutexes.
In our first solution, we used some booleans for the checks, but we had the risk of busy waiting and some shared variables were not protected with the mutexes accordingly.
Therefore, after refactoring our initial solution, we finally made sure that there is no busy waiting, shared variables are protected and we avoid getting deadlock and starvation.

```
# Santa Clause
from Environment import *

elves = MyInt(0, "elves")
reindeer = MyInt(0, "reindeer")
santaSem = MySemaphore(0, "SantaSemaphore")
reindeerSem = MySemaphore(0, "ReindeerSemaphore")
elvesSem = MySemaphore(0, "elvesSem")
ElfMutex = MyMutex("ElfMutex")
ReindeerMutex = MyMutex("ReindeerMutex")
SantaMutex = MyMutex("SantaMutex")



def santa():
```

```python
    while True:
        santaSem.wait()
        SantaMutex.wait()
        if elves.v >= 3:
            print("helpElves()")
            elvesSem.signal(elves.v)
            elves.v = 0
        elif reindeer.v == 9:
            print("prepareSleigh()")
            reindeerSem.signal(9)
            reindeer.v = 0
        SantaMutex.signal()


def elf():
    while True:
        ElfMutex.wait()
        elves.v += 1
        if elves.v == 3:
            santaSem.signal()
        ElfMutex.signal()
        elvesSem.wait()
        print("getHelp()")


def Reindeer():
    while True:
        ReindeerMutex.wait()
        reindeer.v += 1
        if reindeer.v == 9:
            santaSem.signal()
        ReindeerMutex.signal()
        reindeerSem.wait()
        print("getHitched()")


def setup():
    subscribe_thread(santa)
    for i in range(7):
        subscribe_thread(elf)
    for i in range(9):
        subscribe_thread(Reindeer)
```

Output

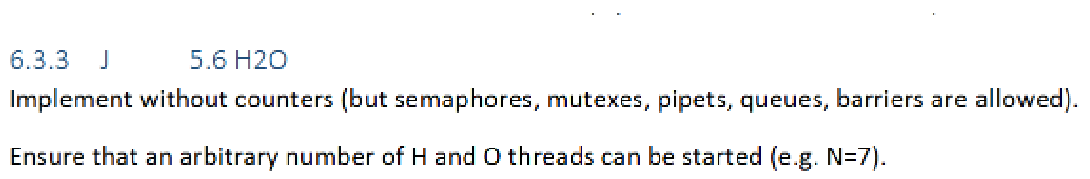


```
helpElves()
getHelp()
getHelp()
getHelp()
helpElves()
getHelp()
getHelp()
getHelp()
getHelp()
helpElves()
getHelp()
getHelp()
getHelp()
getHelp()
helpElves()
getHelp()
getHelp()
getHelp()
getHelp()
helpElves()
getHelp()
getHelp()
getHelp()
prepareSleigh()
getHitched()
getHitched()
getHitched()
getHitched()
getHitched()
getHitched()
```

A video is also included alongside this document.

Third assignment

Problem description



6.3.3 J 5.6 H2O

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

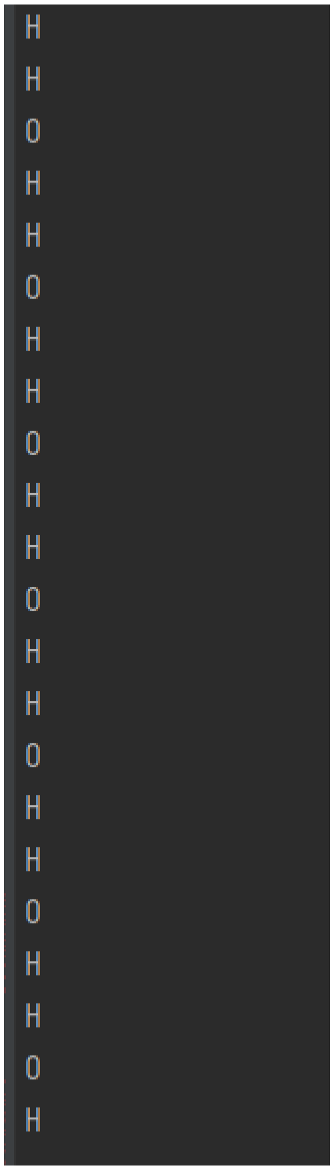
Ensure that an arbitrary number of H and O threads can be started (e.g. N=7).

Solution

In this exercise, we solved this problem using two pipets and two turnstiles (one for oxygen and one for hydrogen). The mutex/pipet is used to control and limit the flow of the threads. Also, we use **hTurnstile** to enter the critical section of Hydrogen and we use **oTurnstile** to only allow oxygen thread to proceed once two hydrogen are present.

```
N = 7
hPipet = MyMutex("hPipet")
oPipet = MyMutex("oPipet")
hTurnstile = MySemaphore(0, "hTurnstile")
oTurnstile = MySemaphore(0, "oTurnstile")

def OxygenThread():
   while True:
       oPipet.wait()

       hTurnstile.signal(2)

       oTurnstile.wait()
       oTurnstile.wait()

       print("O")

       oPipet.signal()


def HydrogenThread():
   while True:
       hPipet.wait()

       hTurnstile.wait()

       print("H")
       oTurnstile.signal()
       hPipet.signal()


def setup():
   for i in range(N):
       subscribe_thread(HydrogenThread)
       subscribe_thread(OxygenThread)
```

Output



```
H
H
O
H
H
O
H
H
O
H
H
O
H
H
O
H
H
O
H
H
O
H
```

A video is also included alongside this document.

Fourth assignment

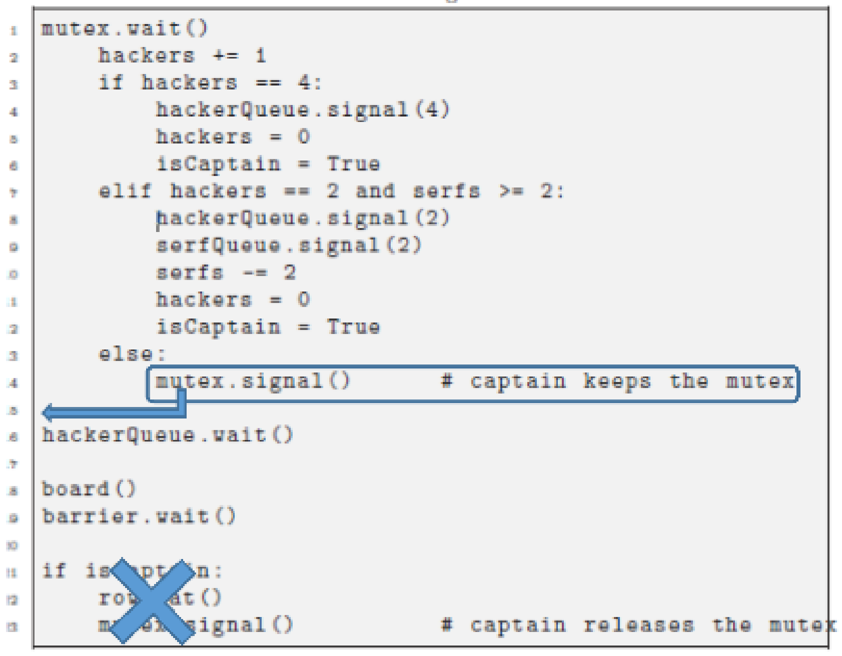
Problem description

6.3.4 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.



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
```

Solution

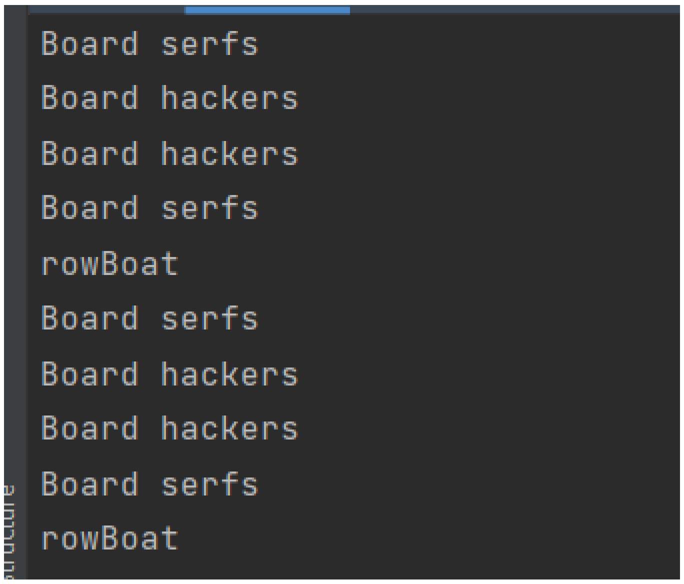
In this exercise, we solved this problem using multiplexes, semaphores and barriers. We use multiplexes to make sure that there is only one boat that can be used any time and the initial value is set 5, because if it is less than 5, deadlock will occur since we can end up having 3 serfs and 1 hacker, or the other way around, which is something that should not occur as it is mentioned in the problem's requirements. We also used barriers to make sure that all the passengers wait for the last passenger and leave at the same time.

After having the feedback session, it was brought to our attention that it would be a better idea to make classes in order to make the code more readable.

```python
from Environment import *

serfSem = MySemaphore(0, "serfSemaphore")
hackerSem = MySemaphore(0, "hackerSemaphore")
barrier = MyBarrier(4, "barrier")
Multiplex = MySemaphore(5, "multiplex")
passengersMutex = MyMutex("passengersMutex")
mutex = MyMutex("mutex")
serfs = MyInt(0, "SerfCounter")
hackers = MyInt(0, "HackerCounter")
passengers = MyInt(0, "PassengerCount")


def RiverCrossingThread(me, other):
   while True:
       Multiplex.wait()
       mutex.wait()

       me.counter += 1
       if me == 4:
           me.sem.signal(4)
           me.counter = 0
       elif me.counter >= 2 and other.counter >= 2:
           me.sem.signal(2)
           other.sem.signal(2)
           me.counter -= 2
           other.counter -= 2

       mutex.signal()
       me.sem.wait()
       print(f"Board {me.name}")

       passengersMutex.wait()
       passengers.v += 1
       if passengers.v == 4:
           print("rowBoat")
           passengers.v = 0
       passengersMutex.signal()

       barrier.wait()
       Multiplex.signal()
```

```python
class Person:
   def __init__(self, counter, sem, name):
       self.counter = counter
       self.sem = sem
       self.name = name


serfsClass = Person(serfs.v, serfSem, "serfs")
hackersClass = Person(hackers.v, hackerSem, "hackers")


def setup():
   for i in range(7):
       subscribe_thread(lambda: RiverCrossingThread(serfsClass,
hackersClass))
       subscribe_thread(lambda: RiverCrossingThread(hackersClass,
serfsClass))
```

Output



```
Board serfs
Board hackers
Board hackers
Board serfs
rowBoat
Board serfs
Board hackers
Board hackers
Board serfs
rowBoat
```

A video is also included alongside this document.