Exercise 1: Synchronisation mechanisms in Java.
Semaphore and monitors.

Hand-out: 19 April
Due: 26 April

The goal of the exercise is to get acquainted with different synchronisation patterns commonly used in real-life applications and see how these patterns can be implemented using the concurrency primitives presented in the lecture. Several properties of concurrent computation, such as mutual exclusion, absence of deadlock and starvation, will be discussed and illustrated with simple examples.

1. Producer – Consumer example
The producer - consumer synchronisation pattern is very common in many applications.

Producer objects (running in parallel) store elements in a shared (bounded) buffer; consumer objects (also running in parallel) consume elements from the buffer.

We will have a look at two different implementations of the producer-consumer scenario in Java: one based on the semaphores and the other based on monitors.

Questions:
1.1. Which implementation provides more guarantees (mutual exclusion, fairness, absence of deadlock)?
1.2. How would you re-implement the shared buffer so that it can be accessed by multiple producers and consumers?
1.3. What mechanism(s) would you like to have in the language in order to solve the problem more efficiently?

2. Dining Philosophers example
The dining philosophers problem is a popular scenario that illustrates the use of shared resources and the need for several guarantees: mutual exclusion, absence of deadlock, absence of starvation.

n philosophers are sitting at the table with a large bowl of spaghetti in the middle and n forks distributed around the table in such way that there is one fork to the left and to the right of each philosopher. The spaghetti is so entangled that each philosopher needs two forks to eat it. A philosopher can only use the forks that are immediately to his left and right. A philosopher repeatedly executes the following sequence of actions: think; take forks from the table; eat; put forks back on the table. Usually, a
We will have a look at different implementations of the dining philosophers example in Java. The examples will illustrate different approaches to solving the problem of deadlock and starvation.

Questions:

1. What real-life synchronisation problems can be “mapped” to the dining philosophers problem?
2. Which implementation provides more guarantees (mutual exclusion, fairness, absence of deadlock) and how are they achieved?
3. What mechanism(s) would you like to have in the language in order to solve the problem more efficiently?

3. Homework: Santa Claus
This nice little problem requires a mixture of non-trivial synchronisation patterns in order to be solved. Therefore it is a perfect homework that will help you survive long and boring rainy evenings. Please try to solve the problem and answer the question below. You can submit your solution (in Java or C#) to concur-course@se.inf.ethz.ch if you want feedback; you can also present it in the next exercise session.

Santa lives in his little house in Jyväskylä; his favourite activity is to have a nap. So he sleeps all the time, except for rare moments when he is awaken by either a group of elves or a group of reindeer. There are 10 elves who live in the area and work in a toy workshop. They produce toys and, once in a while, they run out of ideas and need to ask Santa for help. They can only wake up Santa if they form a group of three. There are also nine reindeer that live in the stable nearby; Santa uses them to deliver toys to kids. The reindeer are as lazy as Santa himself, so for the most of the day they just sleep. When they want to work, they need to form a group of nine (that is all the reindeer must join) and wake up Santa. If they manage to do it, Santa does the delivery round and then goes back to sleep (so do the reindeer). It is important to note that in a situation when there is a group of three elves and the group of nine reindeer trying to wake up Santa at the same time, it is the reindeer that get the priority.

Questions:

1. What synchronisation patterns can you see in the scenario?
2. Is it possible to implement all of them using mutexes? Semaphores? Monitors? A combination of these?
3. Can you think of a more advanced mechanism that would help?
4. How do you implement the priority scheduling (reindeer vs. elves)?

Have fun!