Exercise 1. ***Dining philosophers***

(a) We can reach a deadlock with the algorithm described in the exercise. If all five philosophers take the fork to their left, none of them will be able to take the fork to their right anymore. Since none of the locks is relinquished, a deadlock occurs and all five threads stop making progress.

(b) We could for example assign each fork a number and require that each philosopher only take the forks in increasing number. This would avoid a deadlock because if we get a lock with number i, we are also guaranteed to get the lock with number $i+1$ (anybody who takes fork $i+1$ must have have already had fork i). But by only ordering the locks, it is possible for one philosopher to starve.
To solve the problem of starvation, we could count how many times a philosopher has already eaten. Then, we only try to pick up any forks if both neighboring philosophers have eaten a greater or equal amount of times. A data race on this counter is possible, but it is still guaranteed that a philosopher with a smaller count will be able to eat at some point.

Exercise 2. ***Better than Dijkstra?***

In a C-like language, the lock might look as follows (for thread i):

Algorithm 1 Lock

```
1: procedure LOCK
2:     while true do
3:         b[i] = false
4:         while k ≠ i do
5:             while !b[j] do
6:             k = i                                  ▷ Critical section
7:         b[i] = true                                ▷ Remainder
```

To prove mutual exclusion of the lock, assume by contradiction that both threads are in the critical section. If this is the case and assuming that k is not updated in the critical section, we must have had $k = i$ in Thread 1 as well as $k = j$ in Thread 2 at the time of checking $k \neq [i, j]$. But since we can assume that k is volatile and changes are immediately visible to both threads, this can not be the case.

We now show freedom of starvation of the lock. Again, assume by contradiction that one thread starves. Let Thread 1 be the starving thread. Then, whenever we check whether $k \neq [i, j]$, we must have $k = j$. But as soon as Thread 2 leaves the critical section, it sets $b[i] = true$. This will cause Thread 1 to break the while loop and set $k = i$, thus forcing Thread 2 to enter the outer while loop. Thus, eventually we will have $k = i$, a contradiction.

Exercise 3. ***The Java Memory Model***

(a) **Transitive closure:** The transitive closure tells us which airports are reachable from some given airport. Expressed as a table:

From/To	Aachen	Bern	Chemnitz	Dresden	Erfurt	Frankfurt	St. Gallen	Hamburg
Aachen	X	X	X	X			X	
Bern	X	X	X	X			X	
Chemnitz	X	X	X	X			X	
Dresden	X	X	X	X			X	
Erfurt					X			
Frankfurt						X		
St. Gallen	X	X	X	X			X	
Hamburg	X	X	X	X			X	X

(b) **Program order:** The following pairs are in program order:
$(S1, S3)$, $(S1, S4)$, $(S2, S4)$, $(S3, S4)$, $(S1, S5)$, $(S2, S5)$, $(S3, S5)$

(c) **Synchronization Actions** The following items are Synchronization actions:

- a, b depending on the lock (true for monitor lock)
- c, d
- g can be, but is in general not. Writing to the screen might just write to a buffer. It could also cause a syscall. Even if so, context switches in Java are not guaranteed to be synchronization actions.

e, f are never synchronization actions.

(d) **Correctly synchronized?** The first program is not correctly synchronized. $y = r1$ and $x = r2$ don't have a synchronized relationship. Regardless of that, the program will always result in $x = y = 0$. We could for example make y and x volatile, depending on what the program is supposed to do.
As for the second program: $d1$ will have value 1 in the end. The order in the writer thread does in fact matter. Since x is volatile, $a = 1$ has to happen before $x = 1$, which is important for the reader thread. The reader thread is thus guaranteed to read $a = 1$, since there is a synchronization order between the while loop and $x = 1$, which means that $a = 1$ must have happened before reading $a = 0$.

(e) **Testing for synchronization:** This does not imply that the program is correclty synchronized. There are various implementations of the JVM and the program is not necessarily to be synchronized on other JVMs just because "we ran it many times" on one JVM. In general, we should be able to assume that all JVMs implement the Java Language Specification properly, so assuming our program behaves in accordance with the Java Memory Model, it is likely ok.