Homework 1: Written Problems

Problem 1: Exercise 2.17, page 86
Consider the following program:
co <await (x >= 3) x = x - 3;>
// <await (x >= 2) x = x - 2;>
// <await (x == 1) x = x + 5;>
oc
a.) For what initial value(s) of **x** is the program guaranteed to terminate, assuming scheduling is weakly fair? What are the corresponding final values? Explain.
"Guaranteed to terminate" means that all three arms of the **co** will complete.

Weakly fair: A condition becomes true and remains true at least until after the conditional atomic action has been executed

Therefore, we look at the three arms executing in any permutation.

If x = 1, arm 3 will execute, and arm 2 and arm 1 can execute, or arm 1 and arm 2 can execute. **The final values will be 1 in both cases**, so this will **terminate no matter what**.

If x = 6, arm 1 can execute, followed by arm 2 and arm 3, OR arm 2 can execute, followed by arm 1 and arm 3. **Final value in both cases is 6**, so this will terminate no matter what.

b.) Are there additional initial value(s) of **x** for which the program might terminate? If so, show how those values might result in termination, and the final values that might result. Explain.
"Might terminate" means that there is at least one scenario in which all three arms of the **co** will complete, but they are not guaranteed to do so.

If x = 3:

arm 2 can execute, followed by arm 3 and arm 1. Final value = 3

arm 1 can execute, but it won't complete

If x= 4:

Arm 2 can execute, but it won't complete

Arm 1 can execute, followed by arm 3 and arm 2. Final value = 4

Problem 2: Exercise 2.19, page 86
Consider the following program:

```
int x = 10, y = 0;
co while (x != y) {x = x - 1; y = y + 1;}
// <await (x == y);> x = 8; y = 2;
oc
```

For this program to terminate, both arms of the **co** statement must finish execution; i.e., if one or both arms never completes execution, then the program does not terminate.

a.) Explain what has to happen for the program to terminate. When it does terminate, what are the possible final values for **x** and **y**? If there is more than one possible set of **x** and **y** values, explain how all of them can happen.

For the program to terminate, since x is initially set to 10 and y to 0, we have to wait for the while loop to continually loop until x and y are both equal to 5. Finally, we execute arm 2, with the final values for x and y being 8 and 2 respectively.

I don't think there's another set of x and y values.

b.) Are there circumstances where the program does not terminate? If so, describe one example.

Let's say the program's loads and stores are structured in a way that x and y are never loaded in as being equal at the same time. If the equality of the first arm passes (x == y so the program breaks out of the loop), but stores happen after that step and before the second arm, the second arm will never execute, and the program will not terminate. Note: this can't happen if scheduling is weakly or strongly fair).

Problem 3: Exercise 2.20, pages 86-87
(Five of the parts)
Let **a[1:m]** and **b[1:n]** be integer arrays, **m > 0** and **n > 0**. Write predicates to express the following properties.
a.) All elements of **a** are less than all elements of **b**.
c.) It is not the case that both **a** and **b** contain zeros.
d.) The values in **b** are the same as the values in **a**, except they are in reverse order. (Assume for this part that **m == n**.)
e.) Every element of **a** is an element of **b**.
f.) Some element of **a** is larger than some element of **b**, and vice versa.
As an example, here is a solution to part b.:

b.) Either **a** or **b** contains a single zero, but not both.
Solution:
$((\exists i: 1 \le i \le m : a[i] = 0) \wedge (\forall j : 1 \le j \le m, i \ne j : a[j] \ne 0) \wedge (\forall j : 1 \le j \le n : b[j] \ne 0)) \vee ((\exists i : 1 \le i \le n : b[i] = 0) \wedge (\forall j : 1 \le j \le n, i \ne j : b[j] \ne 0) \wedge (\forall j : 1 \le j \le m : a[j] \ne 0))$

a. $(\forall i: 1 \le i \le m: a[i] < (\forall j: 1 \le j \le n: b[j]))$

c. $((\exists i: 1 \le i \le m: a[i] = 0) \wedge (\forall j: 1 \le j \le n: b[j] \ne 0)) \vee ((\exists i: 1 \le i \le n: b[i] = 0) \wedge (\forall j: 1 \le j \le m: a[j] \ne 0))$

d. $(\forall i: 1 \le i \le m: a[i] == b[n - j + 1])$

e. $\forall i: 1 \le i \le m: a[i] \wedge (\exists j: 1 \le j \le n: b[j] == a[i])$

f. $(\exists i: 1 \le i \le m: a[i] > (\exists j: 1 \le j \le n: b[j])) \wedge (\exists k: 1 \le k \le n: b[k] > (\exists l: 1 \le l \le m: a[l]))$

Problem 4: Exercise 2.33
Consider the following program:

```
int x = 10, c = true;
co <await x == 0)>; c = false;
// while (c) <x = x - 1>;
oc
```

a.) Will the program terminate if scheduling is weakly fair? Explain.

Yes, the program will terminate if scheduling is weakly fair because with w.f., every process gets a chance to check the condition and have it hold true until the end of checking.

b.) Will the program terminate if scheduling is strongly fair? Explain.

Because strongly fair requires for the statement to be true infinitely often, I don't think this program will terminate. the second co statement keeps decrementing, and the first co statement only has one number at which it can execute at. S.F doesn't imply that this process will run, so c will remain true and not change, which s.f. can't catch.

Now, add the following as a third arm of the **co** statement:

```
while (c) {if (x < 0) <x = 10>;}
```

I.e., the program would be:

```
int x = 10, c = true;
co <await x == 0)>; c = false;
// while (c) <x = x - 1>;
// while (c) {if (x < 0) <x = 10>;}
oc
```

c.) Will the program terminate if scheduling is weakly fair? Explain.

Yes, the program will terminate if scheduling is weakly fair because when x eventually reaches 0 and the next call is the first co process, the first co process will eventually run and terminate the program.

d.) Will the program terminate if scheduling is strongly fair? Explain.

Yes, because you are resetting x every time it's less than zero, then the await statement will be able to be true an infinitely often amount of times (instead of just once, like in the previous problem). Therefore, s.f. can handle any condition that flips from true to false and back again, meaning this program will terminate eventually.