

Solutions for Exercise Clas 2

1. The final value of x may range from 2 (!) to 10.

Assuming the two processes to be called P_1 and P_2 , this is how it can get as low as 2:

	x
Initially:	0
P_1 reads 0 from x .	0
P_2 increments x four times.	4
P_1 writes 1.	1
P_2 reads 1 from x .	1
P_1 increments x four times.	5
P_2 writes 2.	2

It can be shown (using invariants — not imagination!) that this is the smallest result.

2. First question: NO. $x := x + 2$ and $x := x + 1$ can be executed in any sequential order, but are not atomic.
Second question: NO. $x := 1$ and $x := 2$ are each atomic, but the order is important.
3. The the number of critical references within statements a to f are 2, 1, 1, 2, 1, and 2 respectively. Hence only statements b , c , and e can be considered atomic.
4. If a variable spans more than one memory word, it has to be accessed using several bus cycles. If these words are accessed by other processors or devices, intermediate memory states may be seen. Even if used only by a single processor, the access to a larger memory area (e.g. a record/structure) is likely to be divided into interruptable steps.
5. Usually the least addressable unit of memory is a byte. Thus to change a boolean variable represented as a bit of a byte, it is necessary to read the whole byte into a register, change the bit by masking and finally store the byte again. This will not be atomic.

6. Solution for Share.2

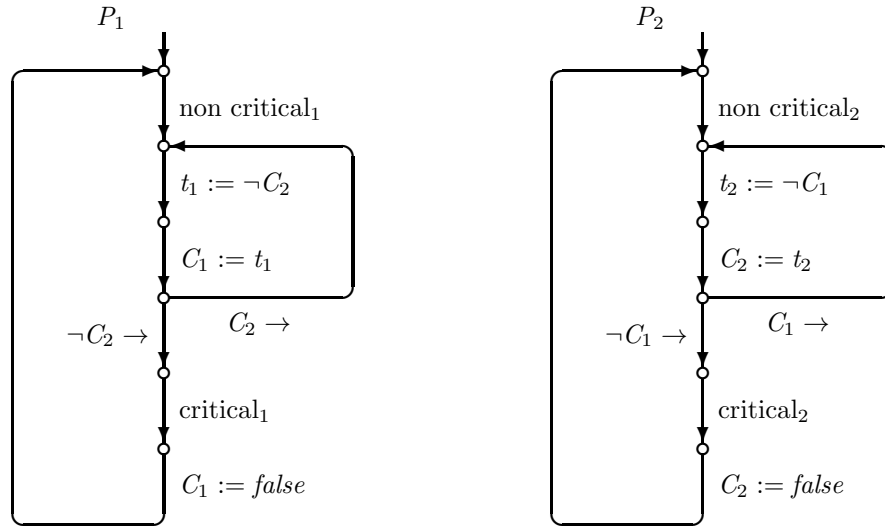
- (a) First we note that the statement $C_1 := \neg C_2$ cannot be considered atomic since C_1 is a shared variable and C_2 is a variable read by the other process. Rewriting to atomic actions we the following entry protocol for P_1 :

```

repeat
  <  $t_1 := \neg C_2$  >;
  <  $C_1 := t_1$  >;
until <  $\neg C_2$  >;

```

Correspondingly for process P_2 . Transition diagrams:



- (b) The algorithm *does not* ensure mutual exclusion. We now see that with the initializations given, an execution in which the atomic actions of the two processes alternate will first set both C -s to true and in the next repetition, both variables false after which both processes will enter their critical section!
- (c) Since the idea of the algorithm is to set ones flag to the opposite of the flag of the other process, it is tempting to believe that the algorithm will work, if the statements $C_1 := \neg C_2$ and $C_2 := \neg C_1$ are executed atomically. But even assuming these to be atomic, the following execution is possible:

	C_1	C_2
Initially:	false	false
P_2 executes nc_2 , its entry-protocol and enters cs_2 .	false	true
P_1 executes nc_1 and (atomically) sets $C_1 := \neg C_2$.	false	true
P_2 leaves cs_2 and executes $C_2 := false$.	false	false
P_1 tests C_2 and enters cs_1 .	false	false
P_2 executes nc_2 , enters its entry-protocol, sets C_2 to true, finds C_1 to be false and enters cs_2 .	false	true

Both processes are now in their critical sections!

The trouble is that the value of C_2 that is tested is not the same as the one that C_1 is set relative to (and vice versa).

[If the **until**-test in P_1 is changed to C_1 and correspondingly in P_2 to C_2 , the algorithm ensures mutual exclusion given atomic assignments.

To actually prove this we need the following auxiliary invariants:

$$\begin{aligned}
 G_i &\triangleq \text{in } cs_i \Rightarrow C_i & i = 1, 2 \\
 I &\triangleq \neg(C_1 \wedge C_2)
 \end{aligned}$$

Now assume that both processes are in their critical sections

$$in\ cs_1 \wedge in\ cs_2$$

According to G_i this would mean that both C variables were true. This, however, would be in contradiction with I . Thus, if G_i and I are invariants, mutual exclusion is ensured.

We are now going to show the auxiliary invariants. G_1 and G_2 are seen to be local invariants.

I is shown by an inductive argument:

- Initially I holds since both C_1 and C_2 are false.
- Since e_1 obviously preserves I , the only potentially dangerous action in P_2 is a_1 :
 a_1 : This actions will preserve I , as C_1 is set to the negation of C_2 and hence one of them will be false after the action.
- By symmetry, all actions in P_2 will also preserve I .

Thus I is an invariant of the program.]

7. In this course, we define a *critical region* to comprise a set of *critical sections* which are pieces of code among which there must be mutual exclusion. In the literature, this distinction is not always made.
8. Yes. The only constraint is that there cannot be two processes active within the same region at the same time.
9. Yes. For instance there may be a region protecting the use of a printer and a region protecting some shared variables. Critical regions may even overlap.
10. (a) $\Box \neg Snows(Bermuda)$
 (b) $\Box (Snows(Helsinki) \Rightarrow Snows(Finland))$
 (c) $\Box (Snows(Norway) \Rightarrow \Diamond Snows(Sweden))$
 (d) $\Box \Diamond Snows(DK) \wedge \Box \Diamond Snows(NZ) \wedge \Box \neg (Snows(DK) \wedge Snows(NZ))$
 (e) $\Box (Snows(Sahara) \Rightarrow \Box Snows(Sahara))$
 (f) $\Box \exists x : Snows(x)$