Exercises by Francesco Zappa-Nardelli, solutions

1 Peterson

It is worthwile to simplify the program a bit, focusing on concurrent execution if the "lock" idiom and getting rid of loops:

```
*flag0 = 1 ;                          *flag1 = 1 ;
*turn = 1 ;                           *turn = 0 ;
if (*flag1 == 0 || *turn == 0) {      if (*flag0 == 0 || *turn == 1) {
  // Enter critical section             // Enter critical section
  ...                                   ...
}                                     }
```

All variables are shared and initialised to zero.

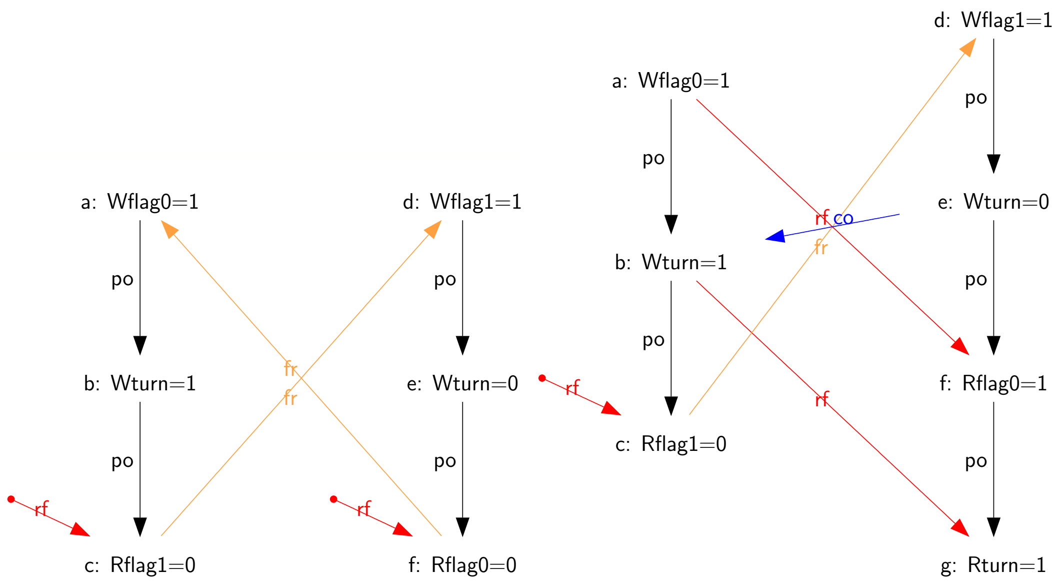
Given the usual left-to-right so called "lazy" semantics of the boolean binary operators, when the left thread is in critical section, it has either read value `0` in `flag1`, or read `1` in `flag1` and `0` in `turn`. Hence considering the two threads we have the following three cases:

1. Thread 0 reads `0` from `flag1`, while thread 1 reads `0` from `flag0`.
2. Thread 0 reads `0` from `flag1`, while thread 1 reads `1` from `flag0` and `1` from `turn`. And the symmetrical case, where thread 1 reads `0` from `flag0`, while thread 0 reads `1` from `flag1` and `0` from `turn`.
3. Both threads read `1` from the other thread flag, and both threads read their identifier in `turn`.

We study those three possibilities, showing that each necessarily leads to a violation of SC.

- When both threads read the initial value of the other thread flag, there are `fr` arrows from each flag read event to the other thread write event. As a consequence we witness a violation of SC of the "SB" kind. See figure 1, left diagram.
- This second case is depicted by the right diagram of figure 1. As thread 0 reads the initial value of `flag1`, there is a `fr` arrow from event c:Rflag1=0 to the write d:Wflag1=1 by thread 1. Furthermore, as thread 1 reads `1` from `turn` *after* having written `0`, the write e:Wturn=0 by thread 1 precedes the write b:Wturn=1 by thread 0. Finally, we have a R-shape cycle: `c -fr-> d -po-> e -co-> a -po-> c`.
- The last case is interesting, as it leads to a quite immediate violation of coherence. Figure 2 depicts the situation where the coherence for `turn` is from thread 1 write to thread 0 write. As a result we have a contradiction between `po` and `fr` on thread 0. Should the coherence order on `turn` be reversed, the contradiction would be on thread 1 between h:Rturn=1 and f:Wturn=0.

Figure 1: Violation of SC, SB and R shapes.



If Peterson code is executed on a TSO machine, tests SB and R are allowed, as a result mutual exclusion will fail. Test SB (left diagram of figure 1) can be forbidden by inserting a fence between events a and c on the one hand and between events d and f on the other hand. Test R (right diagram of figure 1) is forbidden by inserting a fence between the write b and the read c. While forbidding the symmetric situation calls for executing a fence between the write to `turn` and the read of `flag0` by thread 1. It is interesting to notice that case 3. does not call for any fence, as any (reasonable) machine will reject this behaviour.

Finally, inserting fence instructions immediately after the write to `turn` on both threads will suffice for Peterson code to function properly, guaranteeing mutual exclusion. Notice that following the simple rule of executing a fence whenever a read to a shared variable follows a write to a different shared variable would insert the same fence instructions.

2 RWC (Read to Write Causality)

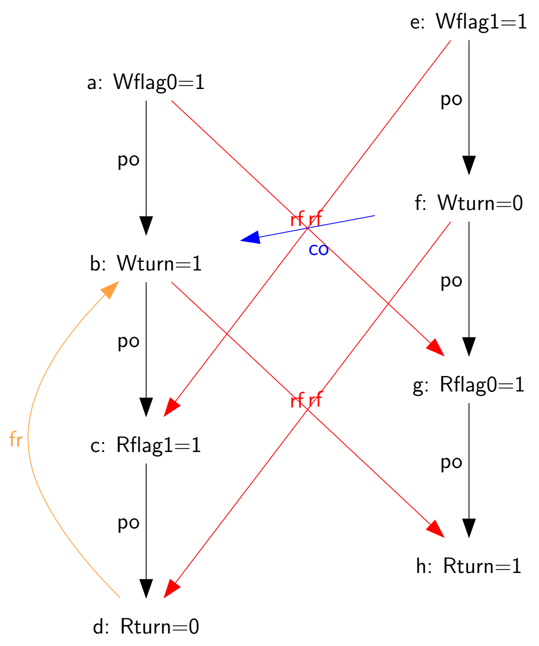
Let write the test in C:

```
void P0(int *x) {
  *x = 1 ;
}

void P1(int *x,int *y) {
  int r0 = *x;
  int r1 = *y;
}

void P2(int *y,int *x) {
  *y = 1 ;
```

Figure 2: Violation of coherence



```
   int r0 = *x;
}
exists (1:r0=1 /\ 1:r1=0 /\ 2:r0=0)
```

The final values of registers forces `fr` and `rf` arrows. (a) The cycle in figure 3 proves that the behaviour is not SC. However (b) it is allowed by TSO as `po` from write to read is not part of `ppo` (figure 4). For (c) adding fence instructions in the middle of thread 1 and 2 will forbid the behaviour. In effect, strong fences `sync` are necessary.

Figure 3: Read to Write Causality.

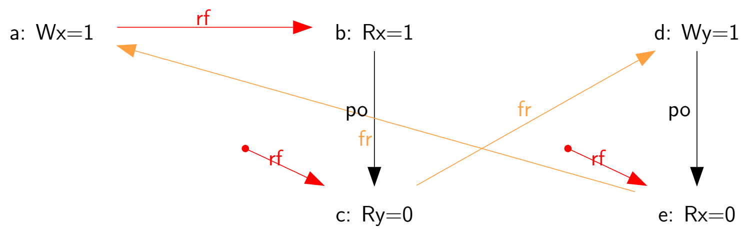


Figure 4: TSO model (excerpt)

```
TSO

include "x86fences.cat"
include "cos.cat"

irreflexive po;(rf|fr|co)+ as uniproc
...
#ppo
let ppo = [R];po;[R] | [M];po;[W] | [M];po;[MFENCE];po;[M]

let ghb = ppo | rfe | fr | co

acyclic ghb as tso
```

3 Data race freedom

(a) race on `y`, (b) DRF, (c) race on `x`.

4 DRF Semantics

All programs can output 42. First (a) can normally output 42. Programs (b) and (c) are racy can thus have any behaviour and thus can output 42…

5 Valid compiler optimisations

(a) and (b) are invalid. For (a) consider test MP: after transformation there is an additional outcome: `1:r0=1; 1:r1=0;`.

```
C MP
{}

P0 (int* y,int* x) {
  *x = 1;
  *y = 1;
}

P1 (int* y,int* x) {
  int r0 = *y;
  int r1 = *x;
}

~exists (1:r0=1 /\ 1:r1=0)
```

```
C TR-MP
{}

P0 (int* y,int* x) {
  *x = 1;
  *y = 1;
}

P1 (int* y,int* x) {
  int r1 = *x;
  int r0 = *y;
}

exists (1:r0=1 /\ 1:r1=0)
```

For (b) consider SB, applying the transformation twice.

```
C SB
{}

P0 (int* y,int* x) {
  *x = 1;
  int r0 = *y;
}

P1 (int* y,int* x) {
  *y = 1;
  int r0 = *x;
}

~exists (0:r0=0 /\ 1:r0=0)
```

```
C TR-SB
{}

P0 (int* y,int* x) {
  int r0 = *y;
  *x = 1;
}

P1 (int* y,int* x) {
  int r0 = *x;
  *y = 1;
}

exists (0:r0=0 /\ 1:r0=0)
```

Transformation (c) is correct: to any execution of the transformed program once associate an execution of the original program by prefixing read event generated by the second load by a copy of it that would be generated by the elided load.