Exercises – Francesco Zappa Nardelli

1. Peterson algorithm is a classic solution to the *mutual exclusion* problem: in all executions, the instructions of the critical sections of the two threads are not interleaved.

```
flag0 := false;
flag1 := false;

flag0 := true;                          flag1 := true;
turn := 1;                              turn := 0;
while (flag1 && turn = 1);      ||      while (flag0  && turn == 0);
// critical section                     // critical section
   ...                                    ...
// end of critical section              // end of critical section
flag0 := false;                         flag1 := false;
```

   (a) Explain informally why the two threads cannot be inside the critical section at the same time.

   (b) Does Peterson algorithm guarantee mutual exclusion if executed on a multiprocessor machine where store buffers are observable (e.g. x86)? In case, where would you put memory barriers to ensure the correctness of the algorithm?

2. Consider this x86 example. Initially all registers and `[x]` and `[y]` are `0`.

```
Thread 0          Thread 1           Thread 2
MOV [x] <- 1      MOV EAX <- [x]     MOV [y] <- 1
                  MOV EBX <- [y]     MOV ECX <- [x]
```

   Finally: `Thread 1: EAX=1`, `Thread 1: EBX=0`, `Thread 2: ECX=0`.

   (a) Is this allowed with respect to an SC semantics?

   (b) Prove whether or not it is allowed with respect to the x86-TSO abstract machine.

   (c) Rewrite the above program using Power assembler. Is the final state allowed in the Power memory model? If yes, discuss the options to insert memory barriers or other instructions so that this final state is forbidden.

3. Which of the following programs are data race free? Justify your answer by either showing a 'racy' execution or by giving a reason why there cannot be a data race.

   (a)
```
Thread 1: lock m; *x = 1; unlock m; *y = 1
Thread 2: lock m; r = *x; unlock m; if (r = 1) then print *y
```

   (b)
```
Thread 1: *y = 1; lock m; *x = 1; unlock m;
Thread 2: lock m; r = *x; unlock m; if (r = 1) then print *y
```

   (c)
```
Thread 1: *y = 1; lock m; *x = 1; unlock m;
Thread 2: lock m; r = *x; unlock m; if (*x = 1) then print *y
```

   where `m` is a monitor, `x` and `y` shared-memory locations and `r` is a local variable. Assume that all memory locations are zero-initialised.

4. Let us assume that our language has the DRF principle as its memory model. Which of the following programs can output `42`? Why?

   (a)
```
Thread 1: lock m; *x = 1; unlock m
Thread 2: lock m; *x = 2; unlock m
Thread 3: lock m; if (*x = *x) then print 42; unlock m
```

(b)
```
Thread 1: lock m1; *x = 1; unlock m1
Thread 2: lock m2; *x = 2; unlock m2
Thread 3: lock m1; lock m2;
          if (*x = *x) then print 42;
          unlock m2; unlock m1
```

(c)
```
Thread 1: lock m1; *x = 1; unlock m1
Thread 2: lock m2; r = *x; unlock m2;
          if r = 1 then print 1
```

where `m`, `m1`, `m2` are monitors, `x` is a shared-memory location and `r` is a local variable. Assume that all memory locations are zero-initialised.

5. Which of the following program transformations are correct under sequential consistency in any context? For the incorrect ones, give a context and an execution where the transformation introduces a new behaviour. For the correct ones argue how the original program could simulate the transformed one (without going into the details of the simulation relation).

(a) `r1 = *x; r2 = *y  =>  r2 = *y; r1 = *x`

(b) `*x = r1; r2 = *y  =>  r2 = *y; *x = r1`

(c) `r1 = *x; *x = r1  =>  r1 = *x`

(`r1` and `r2` are local variables, `x` and `y` are shared-memory locations).