Oblig 2 IN5170
By Finn Eivind Aasen

The Roller Coaster Problem

```
monitor Roller_Coster{
        Semaphore ride_queue := 0; #C is the capacity for passengers in a car and C < N.
        Semaphore check-in = 1;    # N is the number of passenger threads.
                                   #semaphore queue and check uses the principle
                                   #of countdown-and-lock letting the first x thread through
                                   #wait() where x is the initialised value of the semaphore.
                                   #when signaled the semaphore will release a thread from
                                   #the queue or increase it internal counter by 1, letting the
                                   #next thread pass through wait until the internal count is 0
        cond car_full;             #signaled when count = maximum
        cond riding;               #signaled when car unloads passengers after a ride
        cond unloaded;             #signaled when passenger has left car
        int count :=0, maximum := C;
                                   #invariant: (C < N) ∧ (count = 0⇒empty(check-in) ) ∧
                                   #(empty(riding) =true ⇒ count =0)
procedure takeRide(){
        wait(ride_queue);     #only lets c threads pass
        wait(check-in);       #works like a lock only letting one thread modify count at a time
        if(++count = maximum)
                signal(car_full);
        signal(check-in);

        wait(riding);
        signal(unloaded);
}
procedure load(){
        for(int i =1; i <= maximum;i++)       #signal C thread waking up C tread or waking up x
                signal(ride_queue);           #treads and letting (maximum - x) tread pass
                                              #through wait(ride_queue) where x < maximum and
                                              #ride_queue is empty after signalling.
        wait(car_full);                       #wait for passengers to fill the cart.
        count := 0;                           #reset count
}

procedure unload(){
        for (int j =1;j <= maximum;j++){      #unload passenger one after one to make sure car
                signal(riding);               #does not start loading before the cart is empty.
                wait(unloaded);
        }
```

}

}

Greatest Common Divisor

1.

$$\frac{\{\, Pi \,\}\; Si \;\{\, Qi \,\}\ are\ interference\ free}{\{\, P1 \wedge \ldots \wedge Pn \,\}\ co\ S1 \;||\; \ldots \;||\; Sn\ oc\ \{\, Q1 \wedge \ldots \wedge Qn \,\}}\ Cooc$$

    I(x, y) = (x > 0) ∧ (y > 0) ∧ gcd(x, y) = gcd(m,n)

    {I(x,y}Program{x = gcd(n,m)}

We know from the loop invariant that if the program terminates that the condition
    x ≠ y is false after program execution. That means that x = y

Show co process interference free

Let P1 and Q1 be the pre- and post-conditions for S1 in process 1, and accordingly P2, Q2, and S2, for process 2, etc.

We have:

I1 : { P1 ∧ P2 } S2 { P1 }	I3 : { Q1 ∧ P2 } S2 { Q1 }
I2 : { P2 ∧ P1 } S1 { P2 }	I4 : { Q2 ∧ P1 } S1 { Q2 }

Pre-conditions is for S1 and S2 (x >y ∨ x <y ∨ x=y)
Post conditions is for Q1 and Q2 ((x >y ∨ x <y ∨ x=y)

I1: preconditions is ((x >y ∨ x <y ∨ x=y) ∧ (x >y ∨ x <y ∨ x=y)) so simplified to (x >y ∨ x <y ∨ x=y)

    {x >y ∨ x <y ∨ x=y} <if(y>x)y:=y-x;>{x >y ∨ x <y ∨ x=y}

    {(x >y ∨ x <y ∨ x=y) ∧ y>x} ⇒{x >y ∨ x <y ∨ x=y}[y-x/y]
    {(x >y ∨ x <y ∨ x=y) ∧ y>x} y:=y-x{x >y ∨ x <y ∨ x=y}
    {x >y ∨ x <y ∨ x=y} <if(y>x)y:=y-x>{x >y ∨ x <y ∨ x=y}

I2: preconditions is ((x >y ∨ x <y ∨ x=y) ∧ (x >y ∨ x <y ∨ x=y)) so simplified to (x >y ∨ x <y ∨ x=y)

{x >y ∨ x <y ∨ x=y} <if(x>y)x:=x-y;>{x >y ∨ x <y ∨ x=y}

{(x >y ∨ x <y ∨ x=y) ∧ x>y} ⇒{x >y ∨ x <y ∨ x=y}[x-y/x]
{(x >y ∨ x <y ∨ x=y) ∧ x>y} x:=x-y{x >y ∨ x <y ∨ x=y}
{x >y ∨ x <y ∨ x=y} <if(x>y)x:=x-y>{x >y ∨ x <y ∨ x=y}

I3 would look like I1 and I4 would look like I2 because P1,P2,Q1 and Q2 are the same.

1. x >y ⇒ gcd(x, y) = gcd(x − y, y) for x, y > 0
2. gcd(x, y) = gcd(y, x) for x, y > 0
3. gcd(x, x) = x for x > 0

Let P ≡ (x > 0 ∧ y > 0 ∧ gcd(x,y) = gcd(m, n)).
4. {m > 0 ∧ n > 0 ∧ gcd(m, n) = gcd(m, n)} x := m {x > 0 ∧ n > 0 ∧ gcd(x, n) = gcd(m, n)} (Assignment)
5. {m > 0 ∧ n > 0} x := m {x > 0 ∧ n > 0 ∧ gcd(x, n) = gcd(m, n)} (Equivalence)
6. {x > 0 ∧ n > 0 ∧ gcd(x, n) = gcd(m, n)} y := n {P} (Assignment)
7. {x − y > 0 ∧ y > 0 ∧ gcd(x − y, y) = gcd(m, n)} x := x-y {P} (Assignment)
8. {P ∧ (x > y)} x := x-y {P} (Precondition Strengthening)
9. {P ∧ (x ≠ y) ∧ (x > y)} x := x-y {P} (Equivalence)
10. {x > 0 ∧ y − x > 0 ∧ gcd(x, y − x) = gcd(m, n)} y := y-x {P} (Assignment)
11. {x > 0 ∧ y − x > 0 ∧ gcd(y − x, x) = gcd(m, n)} y := y-x {P} (Equivalence)
12. {P ∧ (y > x)} y := y-x {P} (Precondition Strengthening)
13. {P ∧ (x ≠ y) ∧ (x ≤ y)} y := y-x {P} (Equivalence)
14. {P ∧ (x ≠ y)} co S1|| S2 oc {P} (Cooc)
15. {P} while (x≠y) ... {P ∧ (x = y)} (While)
16. {P} while (x≠y) ... {x = gcd(m, n)} (Equivalence)
17. {m > 0 ∧ n > 0} Program {x = gcd(m, n)} (Sequencing)

Because the co part is interference free and the while loop will at some point get x=y as a post-condition from the co-section. Meaning we would exit the while loop and terminate the program.

2.
If we have the while loop inside the co -statement then the outcome would be the same. Each tread would work with the same loop invariant (gcd(x, y) = gcd(n, m)) and instead of for each round in the while loop where the temporary made worker threads would progress one or two steps closer to gcd(m,n), depending on the order of thread execution and the current value of x and y.

If we have the co-statement first then each thread would repeatedly go through the while loop checking if the If statement is true, do work if it is or else do nothing. With the way the if statements are set up, only one of the threads would change the shared variables at a time. The post and pre conditions would be the same too $\{x > y \lor x < y \lor x = y\}$ so i don't think there would be any problem with interference.