Tutorial Session N°1: Mutual Exclusion

Objective: To understand why synchronization is essential in concurrent systems and how to prevent race conditions, deadlocks, and inconsistent results using proper mutual exclusion techniques. And learn how to verify a software solution for mutual exclusion against Dijkstra's four conditions.

EXERCISE 1

Two processes **P1** and **P2** share an integer variable **x**, which is initially equal to **zero (0)**. Each process increments the variable **x** **five (5) times**. The two processes execute their instructions **concurrently without any synchronization mechanisms**.

1. Explain how it is possible that, even though there are ten increment operations in total, the final value of **x** could be **3**.
2. What are all the possible final values of **x** after both processes have finished executing?
3. What is your conclusion?
4. List real-world systems where lack of synchronization can cause serious issues

EXERCISE 2

Study the algorithmic solutions presented below to the mutual exclusion problem. Determine whether each solution is correct, and justify your answer with reference to Dijkstra's four conditions:

1.

```
Common context:
c : boolean; {initialized to false}

Process Pi; {i = 1, 2}
Begin
   ...;
   Repeat
      while c do;
      c := true;
      CRITICAL SECTION;
      c := false;
      ...;
   Until false;
End;
```

2.

```
Common context:
c : boolean; {initialized to false}
Turn : 1..2; {initialized to 1 or 2}

Process Pi; {i = 1, 2}
Begin
   ...;
   Repeat
      Turn := j;
      while (Turn ≠ i) and (c) do;
      c := true;
      CRITICAL SECTION;
      c := false;
      ...;
   Until false;
End;
```

3.

Common context:

Flag: array [0,1] of boolean; {initialized to false}

Turn: 1..2; {Turn initialized to 1 or 2}

```
Process Pi;
Begin
      Repeat
      ::::
      Flag[i] := true
      while (Flag[j]) {
        if (Turn ≠ i) {
           Flag[i] := false;
           while (Turn ≠ i) ;
           Flag[i] := true ;
        }
      }
    CRITICAL SECTION;
      Flag[i] := false;
      Turn := j;
      ::::
      Until false;
End;
```

4.

Common context:

flag: array [1..2] of boolean; {initialized to false}

turn: 1..2; {initialized to 1 or 2}

```
Process Pi; {i = 1, 2}
Begin
  ...
  flag[i] := true;
  while (turn ≠ i) {
     while (flag[j]) do ;
     turn := i;
  }
  CRITICAL SECTION;
  flag[i] := false;
  ...
End;
```

Good Luck