Solutions for 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

SOLUTION

1- How to obtain a final value of x = 3

Each process increments **x** five times. Each incrementation consists of three processor instructions: {Load x; add 1; store x;}. A clock interrupt may occur between any two processor instructions. The two processes execute concurrently (in pseudo-parallel), so several execution sequences are possible:

- Process **P** executes the following sequence five times: { (p0): Load x; (p1): add 1; (p2): store x; } We denote the 5 sequences executed by P as: $[(p0)(p1)(p2)]^5$
- Process **Q** executes the following sequence five times: { (q0): Load x; (q1): add 1; (q2): store x; } We denote the 5 sequences executed by Q as: $[(q0)(q1)(q2)]^5$
- Therefore, the sequence $[(p0)(p1)(p2)]^5[(q0)(q1)(q2)]^5$ represents the sequential execution of the two processes, and the final value of variable **x** is 10.

To obtain a final value of **x = 3**, it is enough for clock interrupts to occur at specific points, producing the following sequence: $(p0)(p1)(p2)(p0)[(q0)(q1)(q2)]^4(p1)(p2)(q0)[(p0)(p1)(p2)]^3(q1)(q2)$

2- The possible values of x after both processes have finished are: $x \in \{2,3,4,5,6,7,8,9,10\}$

The different possible values of **x** can be deduced using the same reasoning as in the previous answer.

3- Conclusion

- Concurrent access to shared data without synchronization causes **race conditions**.
- Results become **non-deterministic** and may be **incorrect** due to lost updates.
- Proper **synchronization** is necessary to ensure correctness when multiple processes modify shared variables.

4 - Real systems where lack of synchronization can cause big problems

- **Banking systems:** Two ATMs withdraw from the same account simultaneously → account can go negative
- **Ticket reservation systems:** Multiple users book the same seat → overbooking
- **Airline reservation / hotel booking websites:** Double bookings if concurrency is not handled
- **Stock trading platforms:** Two traders update the same stock quantity → incorrect trades
- **Multithreaded software / OS kernels:** Shared counters, memory management → crashes or data corruption

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:

Dijkstra's Conditions:

To prove that a solution is correct, it is necessary to show that the four properties stated by Dijkstra are satisfied:

1 - Two processes can never be in their critical sections simultaneously.

2- If no process is in its critical section and a process wishes to enter it, that process must be able to do so within a finite amount of time.

3 - No process executing outside its critical section should block another process.

4 - All processes must share the same solution.

In modern operating systems theory, a **complete *correct solution*** to the critical section problem must satisfy another condition also, called **Fairness (starvation-freedom)**: "***A process requesting access to a shared resource will eventually be granted access; no process can wait forever while others repeatedly proceed***."

So a correct solution must satisfy the 4 Dijkstra's conditions, but a complete solution must satisfy the Fairness condition too.

Solution 1 :

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

Process P1;
Begin
...;
Repeat
while c do;
c := true;
CRITICAL SECTION;
c := false;
...;
Until false;
End;
	Process P2;
	Begin
	...;
	Repeat
	while c do;
	c := true;
	CRITICAL SECTION;
	c := false;
	...;
	Until false;
	End;

If both processes execute while c do; when c = false, they may both pass the while condition simultaneously and each set c:= true. As a result, **both enter the critical section**, violating the first condition. So this is not a correct solution for mutual exclusion.

Solution 2 :

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

Process P1;
Begin
...;
Repeat
Turn := 2;
while (Turn ≠ 1) and (c) do;
c := true;
CRITICAL SECTION;
c := false;
...;
Until false;
End;
	Process Pi; {i = 1, 2}
	Begin
	...;
	Repeat
	Turn := 1;
	while (Turn ≠ 2) and (c) do;
	c := true;
	CRITICAL SECTION;
	c := false;
	...;
	Until false;
	End;

This solution is incorrect, it fails to ensure mutual exclusion (first condition) because both processes can enter the critical section simultaneously when c = false. The use of Turn helps coordinate turns, but because the test and assignment to c are not atomic, the algorithm is unsafe.

Solution 3 : (Dekker solution)

Common context: Flag: array [0,1] of boolean; {initialized to false} Turn: 1..2; {Turn initialized to 1 or 2}	

Process P1;
Begin
Repeat
::::
Flag[1] := true
while (Flag[2]) {
if (Turn ≠ 1) {
Flag[1] := false;
while (Turn ≠ 1) do ;
Flag[1] := true ;
}
}
CRITICAL SECTION;
	Process Pi;
	Begin
	Repeat
	::::
	Flag[2] := true
	while (Flag[1]) {
	if (Turn ≠ 2) {
	Flag[2] := false;
	while (Turn ≠ 2) do;
	Flag[2] := true ;
	}
	}
	CRITICAL SECTION;

Flag[1] := false; Turn := 2; :::: Until false; End;	Flag[2] := false; Turn := 1; :::: Until false; End;

1) Each process raises its flag[i] to signal its intent to enter the critical section. If both processes want to enter simultaneously (flag[0] = flag[1] = true):

- The turn variable decides who goes first.
- The one whose turn it is **not** backs off (flag[i] := false) and waits.

so the first condition is satisfied

2) If one process is outside CS and the other wants in, the waiting process sees the other's flag as false and enters immediately without blocking. So the second condition is satisfied

3) Processes only wait if **the other's flag is true and it's the other's turn**. So, if the other process is outside CS (flag[j] = false, Turn:=j), the waiting process can enters immediately. The third condition is satisfied.

4) and it is clear that the four condition is satisfied

So this is a correct (safe) solution for mutual exclusion.

Also, when the process finishes, it sets turn to the waiting process, ensuring fairness. So, this is a **correct and complete** solution of mutual exclusion. It is the solution of **Dekker (1965)**.

Solution 4 : HOMEWORK

Common context: flag: array[1,2] of boolean; {initialized to false} turn: 1..2; {initialized to 1 or 2}	

Process P1;
Begin
… ;
flag[1] := true;
While turn ≠ 1 do {
While flag[2] do ;
turn := 1;
}
CRITICAL SECTION;
flag[1] := false;
… ;
End;
	Process P2;
	Begin
	… ;
	flag[2] := true;
	While turn ≠ 2 do {
	While flag[1] do ;
	turn := 2;
	}
	CRITICAL SECTION;
	flag[2] := false;
	… ;
	End;

To verify the 1st condition we must take into account all possible execution scenarios of P1 and P2. Consider the following scenario:

turn initialized to 2	

Process P1;
Begin
… ;
flag[1] := true; --------- (1)
While turn ≠ 1 do { ---------- (2) (8)
While flag[2] do ; ---------- (3)
turn := 1; ---------- (7)
}
CRITICAL SECTION; ---------- (9)
flag[1] := false;
… ;
End;
	Process P2;
	Begin
	… ;
	flag[2] := true; ---------- (4)
	While turn ≠ 2 do {---------- (5)
	While flag[1] do ;
	turn := 2;
	}
	CRITICAL SECTION; ---------- (6)
	flag[2] := false;
	… ;
	End;

Both processes are in their critical sections at the same time. Dijkstra's first property is therefore not ensured. So this solution is not a correct solution for mutual exclusion.