CS586: Distributed Computing
TA: Eleftherios Kosmas
Tutorial 2

Exercise 1

Does the following ME algorithm for two processes ensures mutual exclusion and no deadlock? If yes, prove that it is so. If no, present counter-example executions in which these properties are violated. Is starvation of individual processes possible? Prove your claim.

Let $i \in \{0, 1\}$ and $j = 1 - i$ be the identifiers of the processes.
The algorithm makes use of three shared bits $turn$, $flag[0]$ and $flag[1]$.

Code for process i

Initially: $flag[i] = FALSE$ *and* $turn = 0$

```
1:    flag[i] = TRUE;
2:    while (flag[j] == TRUE) {
3:        if (turn == j) {
4:            flag[i] = FALSE;
5:            await(turn == i);
6:            flag[i] = TRUE;
7:        }
8:    }
9:    critical section;
10:   turn = j;
11:   flag[i] = FALSE;
```

Solution:

Notice that the above algorithm is *"one-shot"*, that is each process execute the above code only once.

- **Mutual exclusion**.

  1st Solution: Assume, by the way of contradiction, that both processes 0 and 1 are simultaneously in the critical section at some configuration C. Notice that both process execute line 1 of the pseudo-code before entering the critical section. Let C_0 be the configuration at which process 0 executed line 1 and C_1 be the configuration at which process 1 execute line 1. Assume, without loss of generality, that C_0 precedes C_1. The case where C_1 precedes C_0 is symmetric. Code (lines 1-9) implies that $turn$ does not change value from C_0 to C. We study the following cases according to whether $turn$ equals 0 or 1.

  - $turn = 0$. Then, process 0 does not execute the body of the if statement of line 3, so $flag[0]$ equals TRUE from C_0 to C. Since C_1 follows C_0, process 1 evaluates the condition of the while statement of line 2 to TRUE. Since $turn == 0$, it also executes the body of the if statement (lines 4 to 6) and busy waits on the wait statement of line 5, until $turn$ is updated to 1; this is a contradiction.
  - $turn = 1$. Let C_{cond} be the configuration at which process 0 evaluates the check of line 2 for the first time. If C_{cond} follows C_1, then process 1 evaluates the condition of line 2 to TRUE and this case is symmetric to the previous case. Otherwise, if C_{cond} precedes C_1, then process 0

evaluates the condition of line 2 to FALSE (since initially it holds that $flag[1] = FALSE$) and enters the critical section. So, $flag[0]$ equal TRUE from C_0 to C. Since C_1 follows C_0, process 1 evaluates the condition of the while statement of line 2 to TRUE. Since $turn == 1$, process 1 does not execute the body of the if statement. Therefore, process 1 repeatedly evaluates the while statement of line 2 until $flag[0]$ is updated to FALSE; this is a contradiction.

2nd Solution: Assume, by the way of contradiction, that both processes 0 and 1 are simultaneously in the critical section at some configuration C. Let C_0 be the configuration at which process 0 performs its last write to $flag[0]$ before C and C_1 be the configuration at which process 1 performs its last write to $flag[1]$ before C. Assume, without loss of generality, that C_0 precedes C_1. The case where C_1 precedes C_0 is symmetric.

By the code (lines 1 to 9), it follows that at C it holds that $flag[i]$ = TRUE, $i \in \{0, 1\}$; thus, the value written by process i to $flag[i]$ at C_i is TRUE. Therefore, code (lines 1-11) implies that no process is in its exit section between C_0 and C, and that $turn$ does not change value from C_0 to C. We study the following cases according to whether $turn$ equals 0 or 1.

- $turn = 0$. Process 0 never executes lines 4 to 6 after C_0, so $flag[0]$ equals TRUE from C_0 to C. Since C_1 follows C_0, process 1 evaluates the condition of the while statement of line 2 to TRUE. Since $turn == 0$, it also executes the body of the if statement (lines 4 to 6) and busy waits on the wait statement of line 5, until $turn$ is updated to 1; this is a contradiction.
- $turn = 1$. Let C_{cond} be the configuration at which process 0 evaluates the check of line 2 for the first time. If C_{cond} follows C_1, then process 1 evaluates the condition of line 2 to TRUE and this case is symmetric to the previous case. Otherwise, if C_{cond} precedes C_1, then process 0 evaluates the condition of line 2 to FALSE and enters the critical section. So, $flag[0]$ equal TRUE from C_0 to C. Since C_1 follows C_0, process 1 evaluates the condition of the while statement of line 2 to TRUE. Since $turn == 1$, process 1 does not execute the body of the if statement. Therefore, process 1 repeatedly evaluates the condition of line 2 until $flag[0]$ is updated to FALSE; this is a contradiction.

3rd Solution: Assume, by the way of contradiction, that both processes 0 and 1 are simultaneously in the critical section at some configuration C. Let C_0 be the configuration at which process 0 performs its last write to $flag[0]$ before C and C_1 be the configuration at which process 1 performs its last write to $flag[1]$ before C.

Let $i \in \{0, 1\}$. By the code (lines 1 to 9), it follows that at C it holds that $flag[i]$ = TRUE; thus, the value written by process i to $flag[i]$ at C_i is TRUE. Therefore, code (lines 1-11) implies that process i can not be in its exit section between C_i and C. Furthermore, since process i performs its last write on flag[i] before C at C_i, process i does not execute lines 4 to 6 between C_i and C. Thus, $flag[i]$ equals TRUE between C_i and C.

Assume, without loss of generality, that C_0 precedes C_1. The case where C_1 precedes C_0 is symmetric. Since C_1 follows C_0, process 1 evaluates the condition of the while statement of line 2 to TRUE. Therefore, process 1 repeatedly evaluates the condition of line 2 until $flag[0]$ is updated to FALSE; this is a contradiction.

- **Deadlock**. Assume, by the way of contradiction, that at least one process continuously execute the entry section and no process ever enter the critical section.
  - Assume first that both processes execute the entry section for ever after some configuration C. By the code (lines 1 to 8), it follows that the value of $turn$ does not change after C. Without loss of generality, we assume that $turn = 0$ at C. Then, process 0 can not be stuck on line 5. Thus, process 0 must continuously evaluate the condition of line 2 to $TRUE$. By the code, and since $turn$ equals 0, process 1 will eventually end up waiting forever on line 5. Let C' be the

first configuration after which this holds. Then, at C' and at any configuration after C', $flag[1]$ equals FALSE. This means that process 0 eventually evaluates the condition of line 2 to FALSE and enters the critical section; this is a contradiction.

- Now assume that just one process is blocked in the entry section; without loss of generality assume this is process 0. Since both the critical and the entry sections are bounded, by the code (lines 10 and 11), it follows that that after some point $turn = 0$ and $flag[1] = FALSE$; let C'' be the first configuration at which this holds. By the code (lines 2 to 8), it follows that process 0 can not loop forever in the entry section after C''; this is a contradiction.

- **Lockout**. Assume, by the way of contradiction, that some process (either 0 or 1) continuously execute code of the entry section without ever entering the critical section. Without loss of generality, assume that this is process 0. Since, there is no deadlock in the system and both the critical and the exit sections are bounded, process 1 will eventually execute its exit section. Therefore, after some point it holds that $flag[1]$ equals FALSE and $turn$ equals 0; let C be the first configuration for which this holds. Then, code (lines 1 to 8) implies that after C process 0 will eventually enter the critical section; this is a contradiction.

Exercise 2

Does the following ME algorithm for two processes ensures no deadlock and no lockout? If yes, prove that it is so. If no, present counter-example executions in which these properties are violated.

The algorithm makes use of two shared bits $want[0]$ and $want[1]$, which are initially 0.

Code for process 0	**Code for process 1**
1: while (TRUE) {	8: while (TRUE) {
2: $want[0] = 1$;	9: $want[1] = 0$;
3: await ($want[1] = 0$);	10: await ($want[0] = 0$)
	11: $want[1] = 1$;
	12: if ($want[0] == 1$) then
	13: goto line 9;
4: critical section;	14: critical section;
5: $want[0] = 0$;	15: $want[1] = 0$;
6: remainder section;	16: remainder section;
7: }	17: }

Solution:

- **Deadlock**. Assume, by the way of contradiction, that at least one process continuously execute the entry section and no process ever enter the critical section.

  - Assume first that both processes execute the entry section for ever after some configuration C. By the code (lines 2 and 3), it follows that process 0 can be stuck only on line 3. So, process 0 must continuously evaluate the condition of line 3 to FALSE, after C. Notice that process 0 has already assigned the value 1 to its $want$ bit (on line 2), at some configuration C', and this value does not change at any configuration after C'.

    By the code (lines 9 to 13), it follows that process 1 can be stuck either on line 10 or on the loop of lines 9 to 13 (by continuously executing the goto statement of line 13). Assume that process 1 continuously evaluate the condition of line 3 to FALSE, after C. Notice that process 1 has already assigned the value 0 to its $want$ bit (on line 9), at some configuration C'', and this value does not change at any configuration after C''. This means that process 0 eventually evaluates the condition of line 3 to TRUE and enters the critical section; this is a contradiction.

    Thus, process 1 must continuously execute the goto statement of line 13. Let, C_{goto} be the configuration at which process 1 executes the goto statement of line 13 for the first time. In

order C_{goto} to occur, process 1 must evaluate the condition of line 12 to TRUE; thus, C' precedes C_{goto}. Therefore, code (lines 9 to 10) implies that after C_{goto} process 1 stuck on line 10 for ever. Furthermore, process 1 has already assigned the value 0 to its *want* bit (on line 9), at some configuration C''', and this value does not change at any configuration after C'''. This means that process 0 eventually evaluates the condition of line 3 to TRUE and enters the critical section; this is a contradiction.

- Now assume that just process 0 is blocked in the entry section. Since both the critical and the exit sections are bounded, process 1 will eventually execute its exit section. Therefore, after some point it holds that $want[1]$ equals 0; let C be the first configuration at which this holds. By the code (lines 2 and 3), it follows that process 0 can not loop forever in the entry section after C; this is a contradiction.
- Finally, assume that just process 1 is blocked in the entry section. Since both the critical and the exit sections are bounded, process 0 will eventually execute its exit section. Therefore, after some point it holds that $want[0]$ equals 0; let C be the first configuration at which this holds. By the code (lines 9 to 13), it follows that process 1 can not loop forever in the entry section after C; this is a contradiction.

- **Lockout**. We provide an example showing that the above algorithm does not ensure no lockout property, since process 1 can experience starvation:

  Scenario: (1,2), (9,10), {(3,4,5,6,7,1,2), (10)}*