Summary on Chapter 6.3/6.4 of the book –

6.3– Peterson's Solution

Does not work that well with modern computer because of how modern computers perform basic machine-language instructions. This solution is only here to have an idea of how a solution to this critical section problem could look like. It also illustrates the difficulties of creating a software that will address the requirement of mutual exclusion, progress but also bounded waiting.

Now to prove that this solution is correct. We need to show that:

1. **Mutual exclusion is preserved.**

Processes only enter their critical section if "flag" at position j is equal to false or if "turn" is equal to "i". Two processes cannot enter their critical section at the same time. Because therefore "flag" at position 0 and 1 (for example) will be equal to true. Following these two notes both processes could have not entered their critical section since "turn" cannot be 0 and 1 at the same time. Therefore, one of the processes must have been able to go into their critical section and other one was only able to go after the first left its critical section. This worked because while the first process Pi was going through its critical section "turn == i ". So when the second process Pj arrived it could not go to its critical section because "flag[i]== true" and "turn == i", therefore this is an example of mutual exclusion.

2. **The progress requirement is satisfied / The bounded-waiting requirement is met.**

To prove these two requirements, we can see that P_j is prevented from going to its critical section only if it end up being stuck inside of the while loop with the conditions that "turn == i" and "flag[i] == true". And vice versa, if "turn == j" and "flag[j] == true" and then in this case the Process j will go into its critical section while other processes will have to wait before they can go to their own critical sections. Let's take the example that P_i is in its critical section, inside its while statement it will have two different possibilities. Either set "turn == j" which will make P_j enter its critical section which can also be called progress. Or if "turn == i", then it will make P_i enter its critical section again. We can also see that in the while statement the processes are not changing the "turn" variable. Therefore, if P_j goes into its critical area it will reset the "flag[j] == false" and then process P_i can enter its critical area. But if "flag[j]" is set to true then it will also need to update "turn == j" to be able to go into the while statement again. And because P_i does not change the value of "turn" in its while statement. Therefore, P_i will only enter its critical section after maximum one entry by P_j, and this is called bounded waiting.

6.4 - Synchronization hardware:

All solutions below are based on *locking* (protecting critical regions with the help of locks).

The critical-area problem can be solved simply when it is a single-processor environment. In these conditions we could be sure that two processes cannot run at the same time and therefore nothing will change the variable when they are being used in another process. This is the approach of Nonpreemptive Kernels. However, this approach is not feasible in a multi-processor environment.

Nowadays, modern computers provide us with special hardware instructions that let us test or modify the content of words. These instructions can be used to solve the critical-section problem. The instructions test_and_set() is executed atomically. If two of these instructions are executed at the same time (in two different CPU), they will be executed one after the other in a random order. And this works with the help of the locks. To ensure the mutual exclusion in test_and_set(), we just need to declare a variable "lock" and set it to false. Another hardware instructions is compare_and_swap().

On figure 6.7, we can see that all the requirement for critical-section has been fulfilled. Therefore, to prove the mutual-exclusion requirement we can look at an example. We can see that a process P_i can only enter its critical-section if "waiting[i] == false" or "key == false". Therefore, if another process appears it will be able to go into its critical-section only after the first process is finished and "key" is set to true to allow other process to go into their critical-sections. Then to prove that the progress requirements are met we can look at what is happening in the process P_i when it goes into its critical-section. When this happen the only thing that change is "lock" can be set to false and "waiting[i] == false". Both instructions allow the next process to go into its critical-section. Therefore, the progress requirement is met. Now, bounded-waiting is the last requirement that needs to meet to be able to prove that this is a correct solution to our problem. When, a process leaves its critical-section it scans the "waiting" array by incrementing "i" each round. Then it looks for the first process where the "waiting" is set to true. Therefore, all the process will enter their critical-sections within $n - 1$ turns.