Exam 2

Question 1-

Mutual exclusion:
The algorithm doesn't provide mutual exclusion.
Let's see the next scenario.
Assuming process 0 and 1 are interested by enter in the CS, but 2 isn't.
Let 0 be in the CS, by examining the code we have:
$Write_0$(inter[0] = true) → $Write_0$(TTW = 0) → $Read_0$(inter[2] = false) → CS
Where inter[2] = false by assumption that process 2 isn't interested by enter in the CS, and initialized false.
Let 1 be interested to enter in the CS, then, by examining the code:
$Write_1$(inter[1] = true) → $Write_1$(TTW = 1) → $Read_1$(inter[2] = false) → CS
Where inter[2] = false is by the same assumption that for 0.
We have two processes in the CS, contradicting the mutual exclusion.

Deadlock free:
Assume for the sake of the contradiction that the algorithm doesn't provide deadlock freedom.
Then, by assumption, assuming a set of at least two processes interested by enter in the CS, no one enter in the CS.
If the set is of two processes, we are in the case of the mutual exclusion proof, and we seen than both processes enter in the CS.
Let's now assume that three process are interested by enter in the CS.
By examining the code we have inter[0] = inter[1] = inter[2] = true.
But, TTW is either 0, or, 1, or 2. Then, obviously at least two processes can enter in the CS, contradicting the assumption. The algorithm provide deadlock freedom.

Starvation free:
Assume for the sake of the contradiction that the algorithm doesn't provide starvation freedom.
Then, let's assume that process 0 is stuck forever in the entry code. By examining the code we have:
$Write_0$(inter[0] = true) → $Write_0$(TTW = 1) → $Read_0$(inter[1] = true, inter[2] = true, TTW = 1) → stuck in the while loop.
Where inter[1] = true, inter[2] = true are by assumption that inter 0 is stuck in the while loop.
Then, since inter[1] = true and inter[2] = true, by examining the code we can deduce that process 1 and 2 are also interested by enter in the CS.
Then, both may enter in the CS, since TTW = 0.
In the exit code inter[1] or [2] will be false, then, assuming that 1 and 2 are interested again to enter in the CS, cause if not process o is able to enter in the CS. Then TTW will be either 1 or 2 by examining the code, letting process 0 enter in the CS. Contradicting our assumption, the algorithm provide starvation freedom.

Mutual exclusion new code:
The algorithm doesn't provide mutual exclusion.
Let's see the next scenario.
Assuming process 0 and 1 are interested by enter in the CS, but 2 isn't now.
Let 0 be in the CS, by examining the code we have:
$Write_0$(inter[0] = true) → $Write_0$(TTW = 0) → $Read_0$(inter[1] = false && inter[2] = false) → CS
Where inter[2] = false by assumption that process 2 isn't interested by enter in the CS, and initialized false.
Let 1 be interested to enter in the CS, then, by examining the code:

$Write_1$(inter[1] = true) → $Write_1$(TTW = 1) → $Read_1$(inter[0] = false && inter[2] = false) → Stuck in the while loop
Where inter[2] = false is by the same assumption that for 0.
But now process 2 is interested by enter then, by examining the code we have:
$Write_2$(inter[2] = true) → $Write_2$(TTW = 2) → $Read_2$(inter[0] = false && inter[1] = false) → stuck in the while loop.
But now process 1 is getting running time, and Read that TTW = 2, so he can enter in the CS.
We have two processes in the CS, contradicting the mutual exclusion.

Question 2-

1- The algorithm doesn't provide starvation freedom.
Let's see the next scenario:
Let the process 0 be interested by enter in the CS.
Then, if we have Y != @, the process is sure to return to the line 1 after a finite amount of time.
Also, after the await Y = @, the process is sending in the line 1. But assuming someone else getting running time, he may reinitialized the Y value to another value that @. Then, process 0 will once again enter in the first if. Repeating the scenario we have the process 0 stuck forever in the entry code, contradicting the starvation freedom.

2- Cause the register Y is acting as a semaphore mutex, then, as the mutex algorithm is provide mutual exclusion, we can have a comparison with the register Y. And obviously, there is not link between the register Y and the array inter.

3- The await of the Y = @ is here to give a good chance to the process which was blocked to enter in the CS.
My explanation is that by waiting the Y = @, the process wait that the process in the CS finish his work and enter in the exit code. Then, he may get running time, and with a little chance, can enter in the CS.
Then, we can say that the algorithm don't provide starvation freedom, but with his approach, the chance that one process is stuck forever in the entry code is really small. WRONG ANSWER

4- The same that 3. WRONG ANSWER

Question 3-

Shared: level = 0
shared inter[n] = {false}
shared finish[n] = true

Code for process i.

```
<entry code>
level++
for all level:
    inter[i] = true
    if (i % 2 = 0)
        if(inter[i + 1] = true)
            finish[n] = false
            do: entry_code_peterson(i, i +1)
            finish[n] = true
            await(all k in n, finish[k] = true)
        else
            await(all k in n, finish[k] = true)
    else
        if(inter[i - 1] = true)
            finish[n] = false
            do: entry_code_peterson(i, i -1)
            finish[n] = true
            await(all k in n, finish[k] = true)
        else
            await(all k in n, finish[k] = true)
level = level / 2
if (level != 1)
    goto line 1
<CS>

<exit code>
level--
inter[i] = false
```