Week 1

6.1.1
A: Critical Section

a) In my opinion it is not possible for both threads to enter critical section at the same time.
   - In each scenario this is prevented by variable **lock**, which block current thread on line 6, so second thread could only unlock current thread after completing critical section.
   - Exist scenario where both threads could execute till line 6:

     $$lock[i] = false$$

     In this case both have

     $$flag[i] = true$$

     In such scenario both threads will execute their while cycles until pattern such that:

     $$7A: flag[0] = false$$

     Will be executed just before Thread B check while condition:

     $$6B: while(\ lock[1]\ \&\ flag[1])$$

     Then Thread B exit the while loop, but Thread A will still be inside its own while loop, until Thread B execute critical section and block Thread B and set its **flag** to false.

     $$10B: flag[1] = false$$
     $$11B: lock[1] = false$$

     in other words, allow Thread A to pass waiting loop.

     $$6A: while(\ lock[0]\ \&\ flag[0])$$

     The implementation effectively utilizes locks and flags to enforce mutual exclusion, ensuring that only one thread can access the critical section at a time.

b) Deadlock is possible in such implementation. If During execution both threads goes inside if condition on **Line 3**, So execute **Lines 4,5**.
It is possible when both threads set its **flag** to true on line 1:

$$1A: \text{flag}[0] = \text{true}$$
$$1B: \text{flag}[1] = \text{true}$$

Then threads get to **Line 3** with flags equal to true, then both enter the if condition:

$$3A: \text{if (flag}[1] == \text{true)}$$
$$3B: \text{if (flag}[0] == \text{true)}$$

unlock second thread and set own flag to false telling that it doesn't want to enter critical section so giving second thread permission to pass while cycle

A:

```
4: lock[1] = true;
5: flag[0] = false;
```

B:

```
4: lock[0] = true;
5: flag[1] = false;
```

This happens symmetrical for two threads, so it means that both set corresponding **lock** to true which means both are unlocked.
Then both enter while cycle and will run it forever because it is not possible to change lock inside this cycle, which is part of while condition.

A:

```
while (lock[1] || flag[1])
{
   flag[0] = false;
   flag[0] = true;
}
```

A:

```
while (lock[0] || flag[0])
{
   flag[1] = false;
   flag[1] = true;
}
```

So, deadlock arises without rare condition and most likely to appear when threads are started together.

c) This implementation is quite starvation free, because always after finishing critical section each thread sets **lock** and **flag** to false, so it allows other thread to exit waiting cycle on **Line 6.**

But it might be scenario:

A: Thread A is going through while cycle on **Lines 6-8** with **flag[1]** = true and **lock[1]** = true.

B: Thread B finish critical section and going to execute **Lines 10,11**

Thread A	**flag[0]**	**flag[1]**	**lock[0]**	**lock[1]**	**Thread B**
6	false	true	false	true	9
7	false	false	false	true	10
8	true	false	false	false	11
	true	true	false	false	1
6	true	true	false	false	2
7	false	true	false	true	
	false	true	false	true	3
	false	true	false	true	6
8					9

It appears that **Threads B** after completing cri=tical section gets to its **Line 1** faster than **Thread A** gets to check it's while condition on **Line 6**, then it is possible for **Thread B** to go through **Line 3** and **Line 6** on special State of **flag[0]** while **Thread A** executed **Line 7.**

Such scheduler scenario is relatively rare and in most cases none of the threads will starve, still this implementation has chance of starvation with small probability.

6.1.2
B: Interleaving

It is possible to get 2, here is the scenario:

Step	Description
1A	Thread A load initial value of x (x = 0) to R1 register
2A	Thread A perform inc of R1 so value in R1 is 1, but doesn't save it back to x
1B, 2B, 3B ... 99 times	Thread B fully executes 99 times so value of x is 99
3A	Thread A overwrite value of x with saved in R1 so now x = 1
1B	Thread B loads new value of x (x = 1) to S1 register
1A, 2A, 3A ... 99 times	Thread A fully executes 99 times so value of x is 100
2B	Thread B perform inc of S1 so value in S1 is 2
3B	Thread B save value from S1 to x

So, at the end x is 2

This is only abstract case which shows how it is possible to get 2 at the end, still it is almost impossible to get with any scheduler.

In such problems Semaphores is used to avoid cases where both Thread enters critical section at the same time, so they have access to global variable (save x to its own registers at the same time, so both increment the same value)

In the applications it is more likely to get value from 100 to 200