Question 1:

1. Dekker's algorithm (ME5) was introduced by Dijkstra as the first known correct software implementation for mutual exclusion. Dijkstra introduced the algorithm in stages using ME1-4 as intermediate results. For each of ME1, ME2, ME3 and ME4, explain when it does not work and why.

   Dekker's algorithm is used to share the resource without conflict by using shared memory.
   In this algorithm two threads can share single used resource for communication.

   a) ME 1: In this ME1 if the shared memory/thread is used by using the (turn) process. If the former is running already will be stopped by the process.
   This machine causes a problem with busy waiting. Then the process will run into infinity loop and wait for other process to turn variable which is called live lock.
   This problem is called lock step synchronization. In this process each step will be dependent on other threads for execution.
      - The 'process 0' has started the process and complete the process and will not execute further and the value of turn=0 which never change and the process p1 will be waiting.

   b) ME 2: In this machine two flags will be used one is to represent the current status and the other is to remove lock step synchronization.
   The problem in this machine ME2 is the process will be taken off from the CPU at a certain time, so there is a possibility the both the process 0 and process 1 can enter the critical section at same time.
   In this machine both the process will enter critical section and quit/exit the while condition.
   If the flag[0]= True when the process is updating the it stop the thread and updating will be stopped an exit critical section.
   Then the process 1 finds no one in the critical section then it will enter the critical section and the process 0 will be entering the CPU.

   c) ME 3: In this machine if the flag[0]=flag[1]=true; then deadlock can occur.
   In this machine the flags will not be set before entering the critical section. In this both the process executes the same block of code.
   Int his machine both the process 0 and process 1 set to true this leads to deadlock due to this both the processes can't exit the while loop.

   d) ME 4: In this machine ME4 we have a problem with delay execution. In this we face major problem with bounded wait.
   In this machine we might not have any of the process in the critical section, if any process request to enter the critical section the request my or my not be performed because of block of code(delay). In this case we may also face starvation.
   In this process we cannot calculate the delay of the process which may result in postpone of the delay.

   e) ME 5: In this Deeker's Algorithm complete solution with standing the conditions:
      i. Progress
      ii. Mutual Exclusion
      iii. Bounded Waiting

   In this machine the turn has be set to 0 (turn=0). Both the process 0 and process 1 set there respective flags and access to critical section as well.
   We set the flag[1] = false and wait for turn=1. Then process 0 start execution and change the turn=1 and process 1 enter the critical section and execute.

2. Peterson's algorithm (ME6) is an elegant simplification of Dekker's algorithm. Extend Peterson's algorithm to mutual exclusion among *n (n>2)* processes on shared memory machine.

Process 0:

```
turn = i;

while (flag[i] == true && turn == i)

<enter the critical Section>

<exit the critical section>

flag[j] = (-1)/false;
```

process i

```
flag[i] = true;

turn = j;

<The process is ready to execute and ready to turn>

while (turn == j && flag[j] == true)

<enter the critical Section>

<exit the critical section>

flag[i] = (-1)/false;
```

The peterson's algorithm works totally different than the two process versions shown above.

If the process starts running it start's joining an imaginary queue of process and execute lock().
This process doesn't know exactly how many processes are running or in the queue. In this process we have N total processes. The many goal for this processes is to exit from the lock().
The position for the process 0 is the tail of the queue, the position N-1 is head of the queue, the position N-2 is second from head of the queue.
If in this process the position is -1 then it is not at all in the queue and also not executed the lock().
In this each process looks around at this shared memory information to verify it can be updated and find the exact position of queue. We update the position in two ways:

- If the process comes late and specify that it's in the same position as it thinks it is in the current position at that situation, we update the process.
- If everyone else thinks they're strictly behind, you. Whenever you see one of these situations happen, you can update your estimate of your position to move one step toward the head.

Question 2:

1) This is an example of shared memory using mutual exclusion.
The given algorithm is in FCFS and the is no chance of process fighting for resources. In this only once process can enter critical section based on the priority. There is no point of write or read inconsistency.
The priority based on the assignment of the process to ensure a time that only single process will be executed + concept of CS + the process with higher priority will be read to execute next in critical section.

The process first sets it's 'choosing' variable to be TRUE also indicating to enter CS.
In this each process get's a number, the higher the number has the higher priority or higher the priority it has the higher the number.
In this highest ticket number is assigned to the process corresponds to other processes.
We must make sure that initially the other processes have the 'choosing' variable to FALSE. Which means it has a ticket number and wait for its turn.
In the program the line 1,2, & 3 represent if a process is modifying a ticket then other processes are not allowed to check old/other tickets.

Algorithm: Is FCFS
The process receives a number before entering critical section.
Smallest number holder will enter the critical section.
If processes P(i) and P(j) receive the same ticket number,

```
 If P(i) <P( j)
          P(i) will be served first;
else
          P(j) will be served first.
```

The numbering scheme always generate numbers in increasing order for enumeration: 1, 2, 2, 3, 3, 3, 3, 4, 5….etc.
If the least process enter critical section with the smallest ticket number, no other process will enter the critical section until it executes itself first.

2) The variables are not atomic by using read and write operations. The read operation which is concurrent with a write with same variable can return an arbitrary value.

Example:
We take five processes P0, P1, P2, P3, P4. We set the initial process to 0 for all the processes.

P0	P1	P2	P3	P4
0	0	0	0	0

number[i]:= 1 + max(number[1], number[2], …, number[n]);
P1 gets the value 1 while executing above code.

P0	P1	P2	P3	P4
0	1	0	0	0

The process p3 and p4 get the value 3 because process are not atomic.
number[i]:= 1 + max(number[1], number[2], …, number[n]);

P0	P1	P2	P3	P4
0	1	0	3	3

The process p0 get the value to 4 while executing the line of code.
number[i]:= 1 + max(number[1], number[2], …, number[n]);

P0	P1	P2	P3	P4
4	1	0	3	3

The p1 starts it process and enter critical section because its FCFS priority and execute the process and return the value to 0 after exiting the critical section.

P0	P1	P2	P3	P4
4	0	0	3	3

Then we come to the process p3 & p4 both the process has same value 3 then we entert the process with same ticket number in this section p3 gets a chance to execute first

If processes P(i) and P(j) receive the same ticket number,

```
 If (P(i) <P( j))
        P(i) will be served first;
else
        P(j) will be served first.
```

P3 gets executed and enter critical section and return value to 0

```
k ≠ n ->
        do choosing[k] -> skip od;
        do number[k] ≠ 0 ^ (number[k],k) <lex (number[i],i) -> skip od;
```

P0	P1	P2	P3	P4
4	0	0	0	3

Then p4 gets executed and enter critical section and exit and return the value 0

P0	P1	P2	P3	P4
4	0	0	0	0

The last process P0 enter the critical section and exit and return value to 0

P0	P1	P2	P3	P4
0	0	0	0	0

This algorithm solves the problem with FCFS first come first serve bases. It solves the problem of critical section n(n>2) process

Question 3:

1. In given algorithm the operation flag Z will be true(z[i]:= true;), and the operation of Si will be WRITE operation. The second operation is write to x in Si(x :=i;).
   The operation (x :=i) is WRITE because, the immediately after the first process we will have no intervening operations available for the rest processes.
   The 3rd operation (if y ≠ 0) is a READ operation because, we are specifying them a condition to be check after the 2nd operation so the processer check that statement if (y ≠ 0) then we do

```
z[i] := false;
do y ≠ 0 -> skip od;
        goto start;
```

   The 4th operation (y :=i;) is WRITE operation then we have a if condition (if x ≠ i) is READ operation. Each and every process should perform write followed by read operation in some cases as shown below.

```
y := i;              // write
if x ≠ i ->          // read
z[i] := false;       // write
j := 0;              // write
do j < n ->
do z[j] -> skip od; // write
j := j + 1;          // write
od;
if y ≠ i ->          // read
do y ≠ 0 -> skip od; // read
goto start;
```

   The 5th operation (y := 0;) is WRITE operation and by this process we are stating that the critical section is empty/vacant. The process entering in to critical section will no longer have the contention.

The last operation we make flag Z to false(z[i] := false;) which is WRITE operation. The process left the critical section.
If the last process is write, then the process could not help other to specify to enter or do not enter critical section.
The current process which I explained above have WRITE, WRITE, READ, WRITE, READ, WRITE, WRITE.

In my assumption the absence of contention I assume the sequence to enter critical section is WRITE, WRITE, WRITE, READ, WRITE, WRITE, READ. OR WRITE, WRITE, WRITE, WRITE, READ, WRITE, READ.
The last operation for the process should be read.

2. The above fast_mutex {for process i} is starvation free, because all the process enter the critical section and exited from critical section.
In this lamport's mutual exclusion it uses 8 memory references in the absence of contention.
The given algorithm is faster and reduce the release cost. It also require the additional memory reference in the absence of contention. The given algorithm is deadlock free and block the starvation for every individual process. Lamport's works on completely connected network and it uses FIFO channel for inter process communication. In this each process maintains its owm private request queue for the execution process.

References:

- Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach.* 2nd Ed
- https://books.google.com/books?id=p2XLBQAAQBAJ&pg=PA3&lpg=PA3&dq=Multiple+processes:+The+system+consists+of+more+than+one+sequential+process.+These+processes+can+be+either+system+or+user+processes,+but+each+process+should+have+an+independent+thread+of+control%E2%80%94either+explicit+or+implicit.&source=bl&ots=z1x63-tsGs&sig=ACfU3U3vjVvOP06T8pg0Xlp0deLVOgwpRw&hl=en&sa=X&ved=2ahUKEwiBqtTU4bXlAhUIPq0KHdMLAMoQ6AEwAHoECAgQAQ#v=onepage&q=Multiple%20processes%3A%20The%20system%20consists%20of%20more%20than%20one%20sequential%20process.%20These%20processes%20can%20be%20either%20system%20or%20user%20processes%2C%20but%20each%20process%20should%20have%20an%20independent%20thread%20of%20control%E2%80%94either%20explicit%20or%20implicit.&f=false
- https://books.google.com/books?id=60fSBQAAQBAJ&pg=PA104&lpg=PA104&dq=%7Bprogram+for+clock+i%7D+Define+c%5Bi%5D:+integer+%7Bnon-negative+integer+representing+value+of+clock+i%7D+%7BN(i)+denotes+the+set+of+neighbors+of+clock+i%7D+do+true+%E2%86%92+c%5Bi%5D+:%3D+1+%2B+max%7Bc%5Bj%5D:j+%E2%88%88+N(i)+%E2%88%AA+i%7D+od&source=bl&ots=W6oKtGJgml&sig=ACfU3U3b8hk7lK87wMTlMc3j-rh7WozctA&hl=en&sa=X&ved=2ahUKEwi738yY4rXlAhVMiqwKHfwTBV8Q6AEwAHoECAkQAQ#v=onepage&q=%7Bprogram%20for%20clock%20i%7D%20Define%20c%5Bi%5D%3A%20intege

r%20%7Bnon-negative%20integer%20representing%20value%20of%20clock%20i%7D%20%7BN(i)%20denotes%20the%20set%20of%20neighbors%20of%20clock%20i%7D%20do%20true%20%E2%86%92%20c%5Bi%5D%20%3A%3D%201%20%2B%20max%7Bc%5Bj%5D%3Aj%20%E2%88%88%20N(i)%20%E2%88%AA%20i%7D%20od&f=false

- http://web.cs.iastate.edu/~chaudhur/cs611/Sp09/notes/lec03.pdf
- http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.122.2688
- http://delivery.acm.org/10.1145/10000/7352/p1-lamport.pdf?ip=35.46.53.91&id=7352&acc=ACTIVE%20SERVICE&key=B5D9E165A72B697C%2E724B5EC19F549CA5%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1571949019_d5c62086055d9404af6d61b4b1837cb0
- https://lamport.azurewebsites.net/pubs/fast-mutex.pdf
- https://cs.stackexchange.com/questions/60857/understanding-n-process-petersons-algorithm
- https://lamport.azurewebsites.net/pubs/bakery.pdf