ANSWER 1

a) \( \text{flag}[0] = \text{true}; \text{flag}[1] = \text{false}; \text{turn} = 0. \)

b) If both processes are willing, one process can enter the critical section only once before the other process does so. That means that processes will take turns, they will enter CS alternatively.

Detailed Explanations:

(a) When process 0 enters its critical section, the shared variable \text{flag}[0] is equal to true, since before process 0 enters its CS, \text{flag}[0] has already been set to true (step 0.2). Only process 0 can modify the value of \text{flag}[0] to false and this happens only after process 0 completes its critical section. The value of \text{flag}[1] can either be false or true depending on different scenarios. Assume process 0 and process 1 want to enter their critical sections at the same time and \text{turn}=1. Process 0 will be stuck at step 0.6 waiting the condition of \text{turn}=0 to end the loop. Process 1 grabs the permission to enter its critical section. When process 1 completes its CS and sets \text{flag}[1]=false, and \text{turn}=0 (step 1.11), process 0 can end the loop in step 0.6 and enter its critical section with \text{flag}[1] value of false. In another situation, assume the non-critical section of process 1 doesn’t take much time to execute. After process 0 detects \text{flag}[1]=false and \text{turn}=0, it resets \text{flag}[0]=true and quits the loop at step 0.6. But right before process0 enters its critical section, process 1 has already completed its execution of non critical section codes and goes back to the beginning of loop, setting \text{flag}[1]=true (step 1.2). In this case, process 0 enters its CS with \text{flag}[1] being true. Turn is 0 for sure when process 0 enters its CS. Since process 0 can enter its CS only when \text{flag}[1]=false or \text{turn}=0, which can be satisfied when process 1 executes step 1.11. Since after that, process 1 does not change the value of turn. Process 0 can change turn value at step 0.11. But this happens only after process 0 completes its CS. Therefore, when process 0 enters its CS, \text{turn}=0.

(b) If both processes want to enter their critical sections, in general case, the maximum number of times one process can enter its critical section before the other process does so is 1. Since at the beginning, both \text{flag}[0] and \text{flag}[1] are set to true and \text{turn}=1, Process 1 wins the game and enters its critical section, while process 0 stuck at step 0.6. When process 1 complete its CS and sets \text{flag}[1]=false and \text{turn}=0, process 0 sets \text{flag}[0]=true; this will prevent process 1 to enter the loop at 1.3. This time since \text{turn}=0, process 0 doest not execute the operations inside the “if” structure at step 0.4 and process 1 has to set \text{flag}[1]=false at step 1.5. Therefore process 0 enters its critical section. But in a particular case when one process runs much fast than the other, the number of times for one process to enter its CS continuously can be more than 1. Suppose process 0
is stuck at step 0.6 to waiting for the value of turn=0. Process 1 completes its CS and sets flag[1]=false and turn=0. But before process 0 detects turn=0 and changes flag[0]=true, process 1 has already completed its non-critical section codes and resets flag[1]=true. Since flag[0] is still false, process 1 can enter its critical section again. Note that, a process can enter its critical section for as many times if the other process is not interested in entering its critical section.

**ANSWER 2**

The proposed solution is inspired by Chandy-Lamport algorithm, with the distinction that the snapshot is taken by the visiting agent and saved in its briefcase. The agent starts by taking a snapshot of its home process, and then starts visiting each node. When the agent returns home it will assemble the global snapshot from the states recorded at the individual nodes. For simplicity we assume that the channels have zero capacity so the channel states are irrelevant.

A node is called visited if it has been reached by the agent and unvisited otherwise. While the agent visits the network to collect the snapshot, a message circulating in the system can be one of the four types:

1. from unvisited to unvisited
2. from visited to visited
3. from visited to unvisited
4. from unvisited to visited

Of these, when a message m propagates from a visited node i to an unvisited node j, there is the potential for a causal ordering between the recordings of s(i) and s(j) by the agent. This is because the following causal chain [record s(i) < send m < receive m < record s(j)] will exist. To avoid this situation, we have to ask process j receiving m to save its current state s(j) into a history variable h(j), before accepting m. It is this saved value that the agent will record as the local state of process j.

In order to distinguish between consecutive traversals of the agent, we use a sequence number, SEQ = \{0,1,2\} in the agent briefcase. We need 3 values to distinguish between the current round and the previous round. Before each round, the home increments this value (modulo 3). Each process has 2 variables, seq and agent_seq, both of which are updated to SEQ when the agent visits the node. The variable is needed to recognize messages from visited to unvisited nodes. If the value of agent_seq is appended to every message, then a message m from a visited node i to an unvisited node j is recognized by the condition: agent_seq(i) = agent_seq(j) + 1 mod 3 (i’s reading happens before j’s reading in the same round). To receive such a message, node j sets its seq to -1 (this signals the agent to record the snapshot at j as the state from j’s history, and not as the current node state at the time of visit), updates its agent_seq(j) to agent_seq(i), and saves its current state s(j) into its history variable. The value of seq = -1 propagates to unvisited nodes. This happens every time a node k with seq(k) == -1 sends a message to an unvisited node n. Node n will perform the same sequence of steps as node j. Eventually,
the agent visits j, records the state of j from j’s history, resets seq(j) to SEQ, and deallocates j’s history. Condition seq(j) == agent_seq(j) means that the agent has visited and taken the snapshot at j.

\{agent program – while visiting process i\}

**agent variables**: SEQ, S;

**process variables**: seq, s, agent_seq, h (initially h empty);

if SEQ = agent_seq(i) + 1 mod 3 and seq(i) <> -1 ->
    seq(i) := SEQ; agent-seq(i) := SEQ; S.i := s(i);
\[\] SEQ = agent_seq(i) and seq(i) <> -1 -> skip
\[\] seq(i) = -1 -> seq(i) := SEQ; s(i) := h(i); delete h(i);
fi

NEXT := dfs

\{program for process i\}

do true -
   if message from j: agent_seq(j) = agent_seq(i) + 1 mod 3 ->
       h(i) := s(i); seq(i) = -1; agent_seq(i) := agent_seq(j);
       accept the message;
   fi
execute the next instruction of the application program
od

**ANSWER 3**

Assume two initiators, each running the Chang-Roberts algorithm independently and simultaneously. They create 2 separate partial spanning trees up to the moment when they meet. We assign colors to differentiate between the two subtrees; let’s say we use red and blue. The problem that arises is what happens when the trees meet. They will have to merge, but the merge has to be done on only one edge! So the main problem of the modified joint algorithm is to guarantee that after the join is performed (the trees’ first meeting), no other join operation is attempted by a different pair of nodes. And, of course, the goal being to construct one spanning tree, we will have to reverse the edges of one of the spanning trees, to ensure correct tree structure.

All messages will include sender’s color, to detect the meeting between 2 nodes belonging to the 2 subtrees. The initiators maintain a variable that specifies if the join has
been done. When a red node meets a blue node, each will send a broadcast within their own subtree to ask the corresponding root if they should go ahead and do the join or not. If they get affirmative answer, they finalize the join and broadcast this fact to the entire subtree. This way, the root will know and also other nodes will also know that no new join is to be performed later. One color has to win, let’s say blue. Therefore, all red nodes need to become blue and the edges reversed in the red subtree.

It is difficult to decide if such an approach will provide speedup. The parallel construction of the subtrees might indicate a speedup, but the additional complexity due to the joining process (broadcasted messages, reversal of directed edges) can contributed substantially to the overall complexity. The network topology, the position of the two initiators, the size of the current subtrees (there is no guarantee that we deal with fairly equal sized trees) are factors that influence complexity.