# 1

This algorithm is modeled very closely on the Test-and-Set algorithm presented in the slides. We can see that this will guarantee mutual exclusion by examining the worst case scenario: multiple threads reach the while loop on line 4 at the same time while the shared lock variable (L) appears to be free. These multiple threads all recognize that the lock is greater than 0 so they proceed to enter the DEC operation. However, the DEC operation is an atomic action so only one thread will be allowed to access this at a time. The first thread (thread A) to get in will decrement the value of the shared L variable from 1 (which is the highest value L can take and is the value which denotes the lock is open) to 0 and will return with a value of false. Thus allowing this thread to proceed to the critical section. The next thread (and all subsequent threads that made it to the DEC operation) will decrement the value of the shared lock bringing the value to a number below 0 and will thus return true and not be allowed to enter the critical section. Once thread A has finished its critical section it sets the lock to one (indicating the lock is free) and continues with its tasks. To produce a more cache efficient algorithm I have implemented a strategy similar to the Test-and-Test-and-Set using the lazy OR operator. For a thread to be accessing its critical section the lock must be less than or equal to zero. So, when a thread is checking to see if it is free to enter the critical section it must first make sure L is larger than 0. By adding the first clause in the while loop found on line 4 we ensure that threads are only writing to the shared variable, and thus updating cache less frequently, when they have checked to see if it is possible for them to enter the critical section.

```
1  int L = 1;
2  co [i = 1 to n] {
3    while (something) {
4      while(L<1 || DEC(L)) ; ##check the lock first, then do DEC
5                             ##if the lock appears free
6      critical section;
7      L = 1;                 ##open the lock
8      non-critical;
9    }
10 }
11 oc
```

*Listing 1: Pseudo code solution for critical section problem using the DEC operation.*

# 2

In this version of the DLF algorithm each node will be maintaining the following parameters locally:

- ID number (unique to each node).
- list of IDs of all connected nodes: neighbors[].
- palette of forbidden colors that have been used by its neighbors: usedcolor($v$) and is initially empty.

In shared memory the following will be stored:

- A structure for each node which will contain: the nodes degree [deg($n$)], random value generated each round [rndvalue($n$)], and its proposed color for the round. This will be referenced by the ID of the node.

Similar to the algorithm presented in the paper this algorithm will be working in rounds where in each round the nodes are trying to obtain a color. Ties are broken the same as is discussed in the paper where for neighboring nodes $n_1, n_2 \in V$ the priority of $n_1$ is higher if:

$$\deg(\mathrm{ID}(n_1)) > \deg(\mathrm{ID}(n_2))$$

or

$$(\deg(\mathrm{ID}(n_1)) == \deg(\mathrm{ID}(n_2))) \text{ and } (\mathrm{rndvalue}(\mathrm{ID}(n_1)) > \mathrm{rndvalue}(\mathrm{ID}(n_2)))$$

Where ID($n_x$) denotes node $x$'s unique ID.

In each round of the algorithm each uncolored node $n$ does the following:

1. Choose a random number uniformly distributed on [0..1] and set the corresponding value in its shared structure.
2. Choose the first legal color (not in node $n$'s forbidden colors list) and set the corresponding element in its structure.
3. At this point there is a synchronization barrier which all processes must reach before continuation.
4. Compare its local parameters to those of each of its neighbors (referenced by neighbors[]) to determine if it has the highest priority.
5. If node $n$ doesn't have the highest priority among its neighbors, or its proposed value clashes with its neighbors proposals update the list usedcolors($n$). Otherwise this node is done and the value stored as its proposed value is its final color.
6. At this point there is a synchronization barrier which all processes must reach before continuation.

The pattern in this algorithm resembles an interacting peers pattern in that everyone is executing the same code but on different portions of the data. There is also some similarity to the interacting peers example discussed in lectures, the Finite Difference Method - Jacobi, where some code is executed followed by a barrier followed by some code then a barrier. Due to the nature of the program several barriers are required to keep the different processes synchronized and accessing the correct information.

If we were to consider the case where we had more nodes in the graph than processors it is clear each processor will have to deal with multiple nodes and this could cause a loss in parallelism in the previously proposed algorithm. In the previous algorithm once a node is colored in it stops working. If we imagine initializing the workload to be even across all processors (in that each processor is handling the same amount of nodes) unless we redistribute the workload as nodes become filled in some processors will inevitably have less work to do than others. In this case it might make more sense to organize this into a variant of a bag of tasks problem with multiple bags and access to each bag is synchronized by the barriers in the algorithm.