1. Is the following a correct solution to the critical section problem? Argue whether or not the alleged solution ensures mutual exclusion, progress, and bounded waiting.

```java
boolean blocked[2] = {false, false};
int turn = 0;

thread T0()
{
    while(true)
    {
        blocked[0] = true;
        while (turn != 0)
        {
            while (blocked[1])
            {
                turn = 0;
            }
        }
        CRITICAL SECTION;
        blocked[0] = false;
        REMAINDER SECTION;
    }
}

thread T1()
{
    while (true)
    {
        blocked[1] = true;
        while (turn != 1)
        {
            while (blocked[0])
            {
                turn = 1;
            }
        }
        CRITICAL SECTION;
        blocked[1] = false;
        REMAINDER SECTION;
    }
}
```

2. Consider our first attempt at a solution to the critical section problem (the algorithm which enforced a rigid alternation of processes). Suppose this algorithm is implemented on a two processor system in which a single memory is shared across a system bus. Simultaneous access to the same memory cell is arbitrated in some random fashion, but each access is atomic. Each process is initiated on its own processor, and the processors have unpredictable speeds. Does the algorithm still provide mutual exclusion?

3. Dekker's algorithm (see next page) was the first correct solution to the 2 process critical section problem. Dekker’s algorithm was argued to provide mutual exclusion while avoiding deadlock, both necessary properties of a correct mutual exclusion algorithm. Is Dekker’s algorithm also starvation-free? (Recall that freedom from starvation means that any process wishing to enter its critical section will be granted access within some finite amount of time (as long as both it and the other process continue to execute instructions). In other words, a process cannot be infinitely “unlucky”; the algorithm ensures eventual access to the critical section.) You may make no assumptions about the relative speeds of the two processes, but you may assume that the code is running on a uniprocessor system and that concurrency is achieved by time multiplexing the CPU.
boolean flag[2] = {false, false};
int turn = 0;

thread T0()
{
while (true)
{
    flag[0] = true;
    while (flag[1])
    {
        if (turn == 1)
        {
            flag[0] = false;
            while (turn == 1) { }
            flag[0] = true;
        }
    }
    CRITICAL SECTION;
    turn = 1;
    flag[0] = false;
    REMAINDER SECTION;
}
}

thread T1()
{
while (true)
{
    flag[1] = true;
    while (flag[0])
    {
        if (turn == 0)
        {
            flag[1] = false;
            while (turn == 0) { }
            flag[1] = true;
        }
    }
    CRITICAL SECTION;
    turn = 0;
    flag[1] = false;
    REMAINDER SECTION;
}
}

4. Suppose you have to implement the Lock/Unlock primitives on a uniprocessor machine which does not support the atomic Test-and-Set and Exchange instructions. Fortunately, there is an atomic instruction called \textit{ijz}, which will increment the contents of a memory location and, if the incremented value is zero, will jump to a specified label. If the incremented value is non-zero, the statement following the \textit{ijz} will be executed. More precisely,

\[
\text{ijz}(m, \text{label}) \equiv \begin{cases} 
    m = m + 1; \\
    \text{if } (m == 0) \text{ goto label;}
\end{cases}
\]

Pseudocode surrounded by angle braces <...> is atomic (i.e., done in "memory-interlock" mode: memory accesses within angle braces are serialized by the memory hardware, even in a multiprocessor system).

Implement \textit{Lock/Unlock} on this machine, using \textit{ijz} and any "conventional" C/C++ statements you may need. Argue for the correctness of your implementation. You may not use interrupt disabling as part of your solution.