ABSTRACT
This is a proposal to refine the solutions presently used for mutual exclusion, especially on systems with multiple processors. Dekker’s algorithm is used as a framework for illustration, however, the idea may be implemented on any algorithm for mutual exclusion.

Keywords: Mutual exclusion, critical section, multiple processors, Dekker’s solution, code scan

1. INTRODUCTION
A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing.[1]
Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion.[1]
Note that mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.
If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).
• No two processes may at the same moment inside their critical sections.
• No assumptions are made about relative speeds of processes or number of CPUs.
• No process should outside its critical section should block other processes.
• No process should wait arbitrary long to enter its critical section.

2. ORIGINAL DEKKER’S ALGORITHM
//flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn = 0 // or 1

P0:
flag[0] = true;
while (flag[1] == true) {
    if (turn ≠ 0) {
        flag[0] = false;
        while (turn ≠ 0) {
            // busy wait
        }
        flag[0] = true;
    }
}

// critical section
...

// remainder section
P1:
flag[1] = true;
while (flag[0] == true) {
    if (turn ≠ 1) {
        flag[1] = false;
        while (turn ≠ 1) {
            // busy wait
        }
        flag[1] = true;
    }
}

// critical section
...
turn = 0;
flag[1] = false;
// remainder section

Processes indicate an intention to enter the critical section which is tested by the outer while loop. If the other process has not flagged intent, the critical section can be entered safely irrespective of the current turn. Mutual exclusion will still be guaranteed as neither process can become critical before setting their flag (implying at least one process will enter the while loop). This also guarantees progress as waiting will not occur on a process which has withdrawn intent to become critical. Alternatively, if the other process's variable was set the while loop is entered and the turn variable will establish who is permitted to become critical. Processes without priority will withdraw their intention to enter the critical section until they are given priority again (the inner while loop). Processes with priority will break from the while loop and enter their critical section.

Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation. Let us see why the last property holds. Suppose p0 is stuck inside the "while flag[1]" loop forever. There is freedom from deadlock, so eventually p1 will proceed to its critical section and set turn = 0 (and the value of turn will remain unchanged as long as p0 doesn't progress). Eventually p0 will break out of the inner "while turn ≠ 0" loop (if it was ever stuck on it). After that it will set flag[0] := true and settle down to waiting for flag[1] to become false (since turn = 0, it will never do the actions in the while loop). The next time p1 tries to enter its critical section, it will be forced to execute the actions in its "while flag[0]" loop. In particular, it will eventually set flag[1] = false and get stuck in the "while turn ≠ 1" loop (since turn remains 0). The next time control passes to p0, it will exit the "while flag[1]" loop and enter its critical section.

If the algorithm were modified by performing the actions in the "while flag[1]" loop without checking if turn = 0, then there is a possibility of starvation. Thus all the steps in the algorithm are necessary.[2]

3. SUGGESTED MODIFICATION

P0:
Int count=1;
flag[0] = true;
while (flag[1] == true) {
    if (turn ≠ 0) {
        flag[0] = false;
        while (turn ≠ 0) {
            count= scan_code(count, P0);
        }
        flag[0] = true;
    }
}

do{
    // critical section
} while( count>0 && priority == 0); // priority is a global variable. Set to 1 when a higher priority //process wants to enter critical section
...

turn = 1;
flag[0] = false;
// remainder section
P1:
    Int count=1;
    flag[1] = true;
    while (flag[0] == true) {
        if (turn != 1) {
            flag[1] = false;
            while (turn != 1) {
                count= scan_code(count, P1);
            }
            flag[1] = true;
        }
    }
    do{
        // critical section
    } while( count>0 && priority == 0);  //priority is a global variable. Set to 1 when a higher priority //process wants to enter critical section
    ...
    turn = 0;
    flag[1] = false;
    // remainder section

scan_code(int count, Process P)
{
    for(int i=0; i< slice_limit; i++)    //slice_limit is the time spent scanning in one iteration.
    {
        If(Read_file(P)==isCritical())
            count++;
    }
}

4. ADVANTAGES:
We scan the code for future references to the critical section. By doing so, we ensure that if there are two references to the critical sections close by, we do not exit the process’ critical section and re-enter. This will save time and also, the longer the process is busy waiting for its turn, the farther it can scan its code; thus, the process is compensated for the waiting time.
The priority flag is used to make sure that if a high priority process enters the waiting queue, the current process is pre-empted and the higher priority queue gets access to the critical section.

5. ACKNOWLEDGMENT
The author would like to thank his professor and mentor, Mr. Krishna Prakash, for his much needed guidance and encouragement throughout the creation of this paper.

References
[1.] http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/mutualExclus

AUTHOR
Mansoor Ahmed is an undergraduate student of Manipal Institute of Technology in the field of Information and Communication Technology. He has done industrial work at Paul Scherrer Institute, Switzerland. He has particular interest in the fields of artificial intelligence, data mining, neural networks and network security.