1) A virtual memory system uses a two-level page table. Suppose each physical memory access takes 12 ns, a TLB access requires 3 ns. To ensure the effective memory access time to be within 17 ns, what would be the minimal TLB hit rate? Assume the multi-level page table walk has negligible overhead other than the memory access times.

   Each physical memory access takes 12 ns
   TLB access requires 3 ns
   Effective memory access time to be within 17ns
   EAT = hit * (TLB + access) + (1 - hit) * (TLB + pgaccess + access)
   17 = hit * (3 + 12) + (1 - hit) * (3 + 12 + 12)
   17 = hit * (15) + (1 - hit) * (27)
   17 = 15hit + (27- 27hit)
   -10 = -12hit
   5/6 = hit
   Minimal TLB hit rate would need to be 5/6.

2)

3)

```
int *lock = 0; // shared variable among processes or threads
entry(int *lock) {
locked = test-and-set-lock(lock);
if (locked) {
//append the process to the waiting process list
add(processId, waitingProcesses);
set state to blocked;
}
}
exit(int *lock) {
*lock = 0;
//select a process with the shortest job from the waiting processes list
nextProcess = selectProcessWithShortestJobFrom(waitingProcesses);
set state of nextProcess to ready;
}
```

The pseudocode for entry and exit subroutines can be used for the mutual exclusion but not necessarily for progress and bounded waiting properties. This is because in the entry subroutine, only one lock can be accessed one at a time. Bounded waiting properties are also not guaranteed because we are adding the process to a waiting list but are setting the state to block meaning that it can wait for an infinite amount of time. For the exit subroutine, we set lock back to 0 which unlocks it so it can be accessed again making it mutually exclusive. The exit subroutine also uses the shortest job algorithm which means progress may not happen due to the fact that some of the longer processes may be stuck in the waiting list. So these subroutines can implement mutual exclusion but maybe not progress and bounded waiting times.

4) Ensure that **readlines thread** only starts to execute after the **completion of the readvocab thread** and **countvocab thread** only starts to execute after the **completion of the readvocab and readlines threads**

```
 //both initialized to 0 because they haven't been done yet
shared semaphore rdvocab = 0;
shared semaphore rdlines = 0;

countvocab_thread() {
        rdvocab.wait(); //waiting for readvocab to be done
        rdlines.wait(); //waiting for readlines to be done
        //then we can start doing countvocab thread
        countvocab();

}
readvocab_thread (...) {
        readvocab();
        rdvocab.signal(); //signaling that readvocab thread has been completed
}
readlines_thread (...) {
        rdvocab.wait(); //waiting for readvocab to be done before doing readlines
        readlines();
        rdlines.signal(); //signaling that readlines thread has been completed
}
```

5)

```
//for implementing the critical region
shared semaphore mutex = 1;

//space in queue
shared semaphore available = BufferSize;

//either Bitcoin or Ethereum
shared variable RequestType;

//items in queue
shared semaphore unconsumed = 0;

shared BufferADT brokerQueue; /* queue*/

void producer () {
        RequestType item;
        while(true) {
                Item = new RequestType();
                //making sure we have room
                available.wait();//down

                //accessing queue
                mutex.wait(); //down
                brokerQueue.insert(item);
                mutex.signal();//up

                unconsumed.signal//signal consumer
        }
}

void consumer() {
        RequestType item;
        while(true) {
                unconsumed.wait(); //block until there is an item to consume

                mutex.wait();
                item = brokerQueue.remove();
                mutex.signal();

                available.signal();
                consume(item);
        }
}
```