CS414 HW 2 Solutions

1. **a**. The system would benefit by having different threads to handle interrupts generated by the different sensors. This would ensure that all events are handled promptly, in a fair manner.
   b. No. It runs on a single processor.
   c. In a situation where each thread flips back and forth very rapidly between the states of being ready to run and being inactive, it might be preferable to have them busy-wait on the ready condition rather than having to save/restore state and move the threads between different queues on each block and unblock.
   d. If there is a significant number of threads that are blocked on a condition, with busy-waiting, this will lead to a huge chunk of the CPU time being wasted (on busy-waiting). This might result in a failure if there are real-time constraints (eg: complete action A within time T)
   e. Yes. The schedule of actions to be performed is accessible by all the threads, and might result in inconsistencies if allowed to be accessed concurrently.

2. a. No: each box is completely independent of all the other boxes.
   b. Yes. The position of one box now affects the action to be taken by other boxes.

3. Yes. Trying to mimic test_and_set with decrement:

   free = 1; (initially)

   ```
   CS_ENTER:
           while(decrement(free)<=0){
                   continue;
           }

   CS_LEAVE:
           free= 1;
   ```

   This gives us Mutual Exclusion and Progress. We can extend this solution, using similar constructs as we did in test_and_set (the waiting[] variables), to ensure bounded waiting is ensured.

4. If the application uses only one kernel thread, the entire process has to wait on a blocking system call (which read() is), i.e., none of the threads can run until the system call returns. If the application uses multiple kernel threads, then when a user thread based on one of the kernel threads makes a blocking system call, threads based on the other kernel threads can still continue running, so the application does not block.
   Linux is a multithreaded kernel so the behavior of the application depends on whether the application uses multiple kernel-level threads or not.

5. (i) It may not be multi-threaded in the first place: the code might be structured in the following manner: Poll for user input for a fixed duration of time, then

complete all animation (based on the user input) at one go, go back to check for user input, and so on. If the animation at any step takes a long time to complete, then any mouse clicks during the animation will take until the animation is completed to get noticed.

(ii) (As one student pointed out) The animation thread might have been given higher priority than the thread that's handling user input.

(iii) (As another student pointed out) Some threads might be deadlocked: each thread is waiting for another to complete, so no progress happens.

6. The following solution interprets "sneaking in" as "getting to enter the Critical Section". If it's instead interpreted as "entering CSEnter and then getting to enter the Critical Section", the answers would change(reduce by 1 each)

Note: Since this is a worst-case scenario, assume P0 has a lower id than does P2.

a. If P0 is now inside CSEnter, and has already picked up a lower number(or the same number), P0 would get to go before P2. In any other case, P2 gets to go earlier. So the answer here is 1.

b. Assuming that P2 has not yet set its number(in the worst case): the worst case would be when P0 is now in the last statement of CSEnter, so that it need not wait for P2 to set its choosing to false. So P0 enters and leaves CS, reenters CSEnter, picks a smaller (or the same) number. Thus P0 would get to go in again. P0 can't enter CS again before P2 does, so the answer here is 2.