**Exercise 1**

1. A safety property asserts that nothing bad happens. Best way to express safety conditions in FSP is with a property keyword. A safety property is an FSP expression that states which actions are allowed; all other actions are assumed to lead an ERROR.
2. Busy-Wait mutex protocol is fair and deadlock free. This is because processes cannot have enter and turn condition true at the same time which means they can’t enter to critical sections at the same time. Yielding turn to other thread is done at the end of the critical section. Eventually, when a process finishes its job at the critical section, it will yield its turn to other thread. So integrity will not break and deadlock will not occur.
3. No. Without synchronization, concurrent threads may interfere on common shared variable. Race condition can occur and integrity of the variable can be compromised.
4. Busy-waiting is not efficient for software processes which wastes cycles as they busy wait. Problem with the semaphore is they can allow several concurrent processes enter the mutex, which can cause problems within the class as multiple instances trying to access shared variables. Monitors are viable choice because they only allow one process to enter the mutex. Also, they could have choose message passing for synchronization.

**Exercise 2**

T1 and T2 are not equivalent because T1 is composed of two processes. Actions that are common between two processes are shared. In T1, c action is shared with two processes. T1 has 4 states. Whereas, T2 is one process by itself, so same actions are not shared. This results T2 to has 5 states.

**Exercise 3**

You open radio and it starts at the highest frequency then you click scan, it locks into a channel in the middle then you click scan again and it reaches to the end of the frequency. From the end, you can click on reset and re-scan.

RADIO = (on -> ON),

ON = (off -> RADIO| scan -> LOCK| reset -> ON),

LOCK = (scan -> BOTTOM| lock -> LOCK| off -> RADIO| reset -> ON),

BOTTOM = (end -> BOTTOM| off -> RADIO| reset -> ON).

**Exercise 4**

const N = 1

Stack = ( push[v:0..N] ->pop[v] ->Stack | push[v:0..N] ->pop[v] ->Stack | push[v:0..N] -> ERROR | pop[v:0..N] -> ERROR ).

Producer = ( push[0] -> Producer | push[1] -> Producer ).

Consumer = ( pop[0] -> Consumer | pop[1] -> Consumer ).

||Process = ( Stack || Producer || Consumer ).

I couldn’t manage to implement slot sizes of stack. Overall, my idea is you can pop only whatever you push (push[v:0..N] ->pop[v] ->Stack), and you can make twice of these because you have two slots. (push[v:0..N] ->pop[v] ->Stack | push[v:0..N] ->pop[v] ->Stack). If you push more than twice,then stack can throw an error (push[v:0..N] -> ERROR) or if you pop before you push, then it can throw another error(pop[v:0..N] -> ERROR).