Read me question 1

In a(1) We are creating a dinings philosopher problem by strict ordering of the philosophers. In the code, we are changing the order of obtaining the left and the right forks for 1 person(ie philosopher 4) who is the last philosopher. This prevents deadlock. We have used 2 semaphores- dining and forks[]. Dining is a binary semaphore (can only take on values 0 or 1) so that only 1 philosopher can access the fork at a time and the forks[] is a counting semaphore for the philosophers that can take on arbitrary non-negative values. Dining acts like a mutex. Sem_wait is used to decrement the value of the semaphore by1. If the value of the semaphore is negative, the calling process blocks and one of the blocked processes wakes up when another process by calling sem_post. Sem_post would then increment the value of the semaphore by 1 and so it wakes up a blocked process that was waiting on the semaphore.

In a(2) We are creating a dinings philosopher problem. The 2 states of the philosophers are thinking and eating. We have used 2 semaphores- dining and forks[]. Dining is so that only 1 philosopher can access the fork at a time and the forks[] is for the philosophers.

In b(1), like a (1) there is strict ordering, but the dining semaphore has been initialized like so: sem_init(&dining, 0, 2) since we have 2 bowls which can be accessed each by the 5 philosophers. The rest of the logic remains the same as in 1(a).

In b(2) like in a(2) We have used 2 semaphores- dining and forks[]. Dining is so that only 1 philosopher can access the fork at a time and the forks[] is for the philosophers. Again the dining semaphore has been initialized like sem_init(&dining, 0, 2) since we have 2 bowls which can be accessed each by the 5 philosophers.