Threads are used to simulate philosophers in all parts. **sleep()** function is called in the eat() function of every program to simulate eating.

In part (a) of the question, 2 variants of the program have been implemented. One is with the strict ordering of resource requests while the other is done using semaphores.

For strict ordering, we are taking resource requests in a fixed order to avoid deadlock. Philosophers 0 to 3 pick up left fork first then right while philosopher 4 picks right first then left. lock_fork() and release_fork() are 2 user defined functions that perform the locking mechanism. lock_fork() checks to make sure that the current fork is released before fulfilling the locking request. It does this through a while loop. The state of forks(locked or unlocked) is tracked using an int array which is initialised with all values equal to -1(released state) and gets set to the philosopher number when locked. The release_fork() function releases the lock on the fork by changing its value in the array to -1.

We have used **semaphores to simulate the forks** and created **threads to simulate the philosophers**. Each semaphore is **initialised at value 1** using **sem_init()** function to ensure that only 1 thread can access it at a given time(i.e., only 1 philosopher holds any given fork at a given time). The threads are created using **pthread_create()** function which executes the void pointer function philosopher() that takes the philosopher number as argument. This function simulates the actions of a philosopher of eating and thinking. **sem_wait()** function is called while accessing the forks. This function decreases the semaphores value by 1 and puts the thread to sleep if the semaphore value is less than 0, otherwise it continues the thread execution. This places a lock on the semaphore and ensures that no other thread can access it. **sem_post()** function increases the semaphore value by 1, thus releasing the lock, and wakes up the threads, if any, that are waiting for execution. **pthread_join()** function has been called on each thread to ensure that they finish execution before program exits.

Part (b) of the question utilises the same concepts as part (a) with the additional change that now we also consider 2 sauce bowls that the philosophers can access at any time(and need 1 for eating).

Part 1 of part (b) has a similar solution to part 1 of part(a). The only changes are the additional functions for picking and keeping the bowls down and the subsequent locking functions which work on the same logic as the ones for forks. We track which philosopher has which bowl at a given time using an array. The get_bowl() function checks whether bowl 0 is acquired(-1 value in the array would mean free while positive values indicate which philosopher has acquired the bowl), and makes a call to lock it if it is free. Otherwise it calls lock on bowl 1.

In part 2 of (b) we have used semaphores to simulate the sauce bowls and initialise them with value 1. The get_bowls() function called in eat() function **sem_getvalue()** on bowl 0 and checks if it is currently being used by another philosopher or not(if semaphore value=1, then

bowl is free at that time, otherwise being used). If bowl 0 is currently in use, it calls **sem_wait()** function on bowl 1 to acquire it, otherwise it calls sem_wait() on bowl 0. An array is used to keep track of which philosopher is using which bowl so that it can be released in put_bowl() function by calling **sem_post()**.