OS Assignment 3 - Q1

1.1 Strict Ordering of Resource Requests

(a) The program uses an array of a custom `struct lock` that essentially does not allow any other thread (philosopher) to access the `lock` variable until it has been set to 1; when the particular `lock` is in use by any thread, it is set to 0 and any thread that needs access to that `lock` waits in a busy-wait loop until the `lock` has been freed (i.e. set to 1 again).

(b) With two sauce bowls, the program uses two arrays of `struct lock` - one to simulate forks and one to simulate the sauce bowls. If any thread has access to the required forks, it waits until either of the two sauce bowls have been freed (set to 1); when in use, the sauce bowl variable is set to 0 by the thread that is accessing it.

1.2 Semaphore

(a) Similar to the strict-ordering procedure, the program uses an array of binary semaphores to simulate the forks. It calls `sem_wait()` that waits for the required forks to be freed (i.e. set to 1). Once the thread has access to these semaphores, it calls `usleep()` to simulate eating for a few seconds. The program then calls `sem_signal()` to free the semaphores (i.e. set them to 1) to be accessed by any other threads that require access to these particular semaphores.

(b) For two sauce bowls, in addition to the binary semaphores that simulate forks, the program uses a 2-counting semaphore i.e. another semaphore that can be accessed by two threads at a time; this simulates, at maximum, two philosophers eating simultaneously. Similar to the binary sempahores, the program calls `sem_wait()` on the 2-counting semaphore before calling `usleep()`, and calls `sem_signal()` after the eating-simulation in order to set the counting semaphore back.