Assignment 4

Aditi Sejal
2019228

I have made a structure called my_semaphore which uses mutexes and conditional variables. To make sure that the critical sections within the semaphores themselves remain protected, I have used mutexes. These are used to lock or unlock a critical section such as incrementing/decrementing semaphore value. Thus phtread_mutex_t type variable is the mutex used and I have used it's pthread_mutex_lock (the mutex object referenced by *it's parameter* shall be locked by calling *pthread_mutex_lock*(). If the mutex is already locked, the calling thread shall block until the mutex becomes available) and pthread_mutex_unlock (the *pthread_mutex_unlock*() function shall release the mutex object referenced by *it's parameter*) functions for the same.
In the blocking variant I have used pthread_cond_t which has functions pthread_cond_wait (atomically unlocks the *mutex* and waits for the condition variable *cond* to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled) and pthread_cond_signal (restarts one of the threads that are waiting on the condition variable, iIf no threads are waiting, nothing happens. If several threads are waiting, exactly one is restarted, but it is not specified which).
In the wait() function, we have acquired the mutex and then decremented the value if it was not already zero to make sure it can not be accessed by other threads and made them wait on it using pthread_cond_wait thus implementing blocking semaphore.
In the signal() function we have acquired the lock, incremented the value of the semaphore, signalled waiting threads and released the mutex.
In the non-blocking variant of wait() and signal() the only change is not using conditional variable and immediately returning from wait and signal if the resource is unavailable. Thus we don't keep the threads blocked.
The signal printer function prints the value of the semaphore.

This semaphore has been used to solve the dining philosopher problem where the 0th philosopher (assuming indexing of both philosophers and forks starts from 0) picks the right fork first followed by the left to avoid a deadlock situation. This alternating of order of picking of forks avoids a deadlock and enables the program to continue infinitely. The number of philosophers has been taken as an input by the user and is by default 5. The 2 rice bowls have been assumed to be a single entity since both are needed by a philosopher to eat. The philosophers are implemented using threads which execute a function which orders them to do the following tasks in a synchronised manner (due to the use of semaphores) - pick forks, pick both rice bowls, eat (here the philosopher id and forks they are using are printed preceded by a call to signal printer to check the value of semaphore), keep forks back, keep both rice bowls back. This continues infinitely as the philosophers think and eat.