Philosophers and Dining problem

Logic Explanation of solution

In this case, there are two ways to solve the problem

1. **Picking up sauce bowls first.**
    a. This is helpful since only a single philosopher would do so and hence only a single philosopher would pick forks. This avoids any possible deadlocks.
2. **Picking forks differently on basis of the index**
    a. If the index is even the left fork is picked first
    b. If the index is odd right fork is picked first
    c. After this, the sauce bowls are picked

I have implemented a mix of both and hence the philosophers try to pick the sauce bowl first and then pick forks differently on the basis of their index.

Implementation details of the solution

1. **My_semaphore includes**
    a. **integer(entry)** used to manage the entry and exit.
    b. **lock(mutex)** used to ensure thread safety of the semaphore(ie to avoid race conditions)
    c. **cond(conditional variable)** used to sleep and wakeup threads and maintain the queue
2. **Blocking implementation includes**
    a. Wait
        i. **pthread_mutex_lock**(waits till the thread is the owner of the lock)
        ii. **pthread_cond_wait**(used to put the thread to sleep if the entry was not allowed to a semaphore and maintains a queue)
        iii. **Pthread_mutex_unlock** used to release the lock so that other threads can use it
    b. Signal
        i. **pthread_mutex_lock**(waits till the thread is the owner of the lock)
        ii. **pthread_cond_signal**(used to wake up a thread waiting for a conditional variable to release)
        iii. **Pthread_mutex_unlock** used to release the lock so that other threads can use it
    c. Signal Print value
        i. Used to print the value of the semaphores

3. **Non-blocking implementation**
    a. **Wait**
        i. **pthread_mutex_trylock**(it tries to take the ownership of the lock and returns a non zero value if it is not able to take the ownership.)
        ii. **Pthread_mutex_unlock** used to release the lock so that other threads can use it
    b. **Signal**
        i. **pthread_mutex_trylock**(it tries to take the ownership of the lock and returns a non zero value if it is not able to take the ownership.)
        ii. **Pthread_mutex_unlock** used to release the lock so that other threads can use it