Concurrent Programming CS511 Assignment 4

Name: LIAN YU

To run the program:

java DiningPhilosophers <k> <l> <h> <e>

The program will keep running until you force it to stop.

1.

In first part of assignment, I implemented the simple solution by using semaphore (semaphore.aquire and semaphore.release). When the eat time is very short (1 million second), and the think time is equal to the eat time, it is the easiest for the program reach deadlock.

It is not difficult to image that when the threads are keeping competing to use the public resources, and the self-run time of the threads are also very short, it is easiest for the threads to reach deadlock.

2.

I implemented an enhancement solution by using treAquire to replace acquire. The tryAquire function will return directly depends on the lock condition, instead of waiting the lock to be released. Thus, the program will never reach deadlock.

My strategy is to make philosophers got two forks before they begin to eat. If they do not successfully get both of the forks, they will put down the already held fork. Then, they will retry to get the forks again. It is busy wait in my code.

I wrote a function which can automatically count the efficiency of every thread, which stop depends on the input target turn. It means when the first thread reach the target turn of the loop, the thread will interrupt all other threads to make them stop to exit. But based on the assignment input usage requirement, I do not invoke the function in my submitted assignment.

The example output of the efficiency of six philosophers, one million second eat, one million second think (which are the same arguments to reach deadlock in first part). And the target turn is 1000:

```
<terminated> DiningPhilosophers (5) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Dec 14, 2012 8:15:34 PM)
Philosopher 0 is eating.
Philosopher 2 is eating.
Philosopher 1 is thinking.
Philosopher 5 is thinking.
Philosopher 3 is thinking.
Philosopher 2 is thinking.
Philosopher 1 is eating.
phi 0 is called with interrupted.
phi 1 is called with interrupted.
phi 3 is called with interrupted.
phi 4 is called with interrupted.
phi 5 is called with interrupted.
philosopher 2 reach target turn
Philosopher 4 is eating.
Philosopher 0 is thinking.
phi 0 ran 917 turns, the sum wait time is 541, efficency is 0.7350054525627044
phi 1 ran 923 turns, the sum wait time is 842, efficency is 0.6890574214517876
phi 2 ran 1000 turns, the sum wait time is 596, efficency is 0.743
phi 3 ran 950 turns, the sum wait time is 873, efficency is 0.6926315789473684
phi 4 ran 918 turns, the sum wait time is 628, efficency is 0.7254901960784313
phi 5 ran 931 turns, the sum wait time is 800, efficency is 0.6928034371643395
code end
```

I think the efficiency is acceptable, because the thread runs in busy wait to try to acquire the public resources. If the eat time is much short than think time, the efficiency will be higher. And the efficiencies of all threads are close. No thread have starvation problem, because when the thread releases the lock, the lock will be acquired so by the other thread.

If I want to improve the efficiency of the program, I think I need to add communication and synchronization features to the program, for example, add check mechanism, before the philosophers try to get the forks, they need to check the state of the fork, which can avoid useless acquire and release operation. Or, add mechanism that after philosophers eat, they will wake up the nearby philosopher.

Furthermore, if it is possible, I can add priority parameter, which can make the threads work in schedule based on priority. Then the program can control the efficiency of threads depend on their priority.