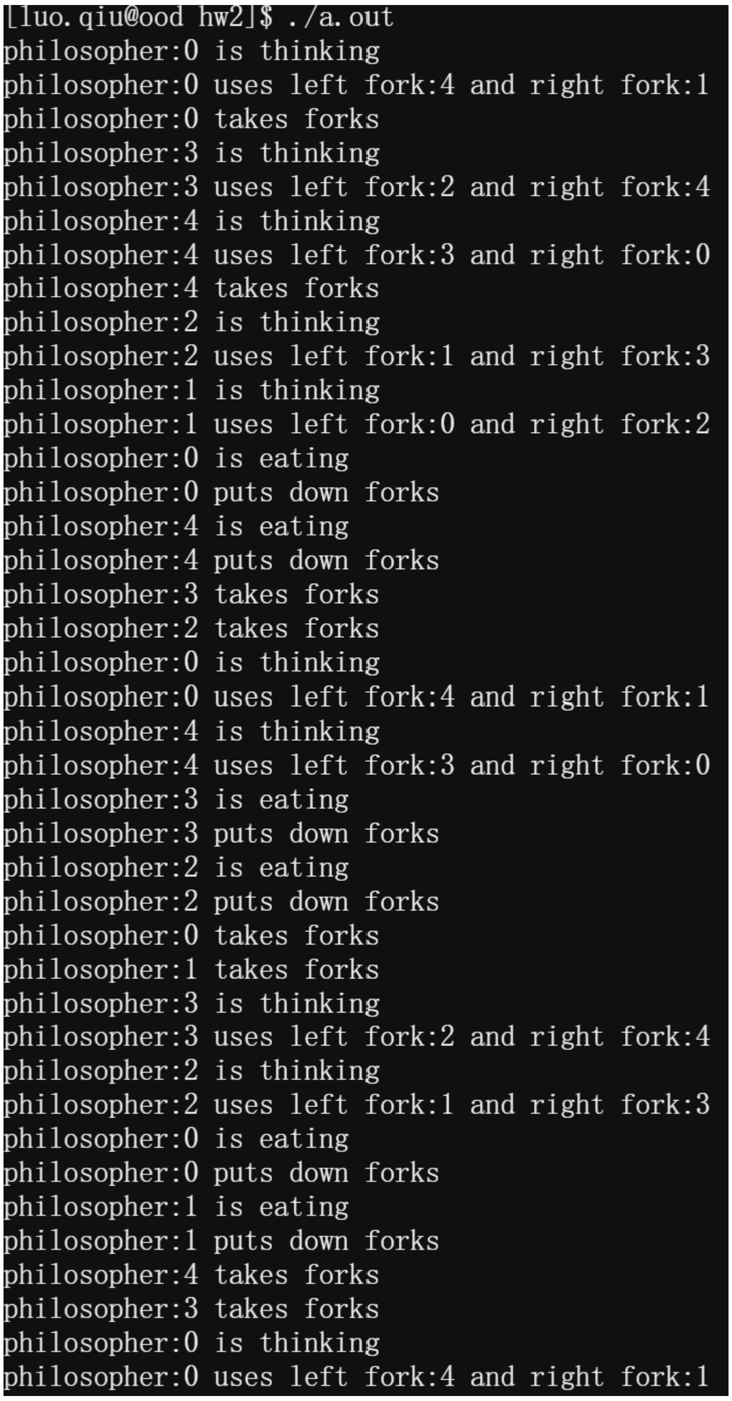
1. In 1965, Edsger W. Dijkstra described the following problem. Five philosophers sit at a round table with bowls of noodles. Forks are placed between each pair of adjacent philosophers. Each philosopher must alternately think or eat. However, a philosopher can only eat noodles when she has both left and right forks. Each fork can be held by only one philosopher, and each fork is picked up sequentially. A philosopher can use the fork only if it is not being used by another philosopher. Eating takes a random amount of time for each philosopher. After she finishes eating, the philosopher needs to put down both forks so they become available to others. A philosopher can take the fork on her right or the one on her left as they become available, but cannot start eating before getting both of them. Eating is not limited by the remaining amounts of noodles or stomach space; an infinite supply and an infinite demand are assumed.

Implement a solution for an unbounded odd number of philosophers, where each philosopher is implemented as a thread, and the forks are the synchronizations needed between them. Develop this threaded program in pthreads. The program takes as an input parameter the number of philosophers. The program needs to print out the state of the table (philosophers and forks) – the format is up to you.

Answer the following questions: you are not required to implement a working solution to the 3 questions below, though adding each case to your C/C++ implementation will earn extra credit (on your quiz grade) for everyone who tries it.

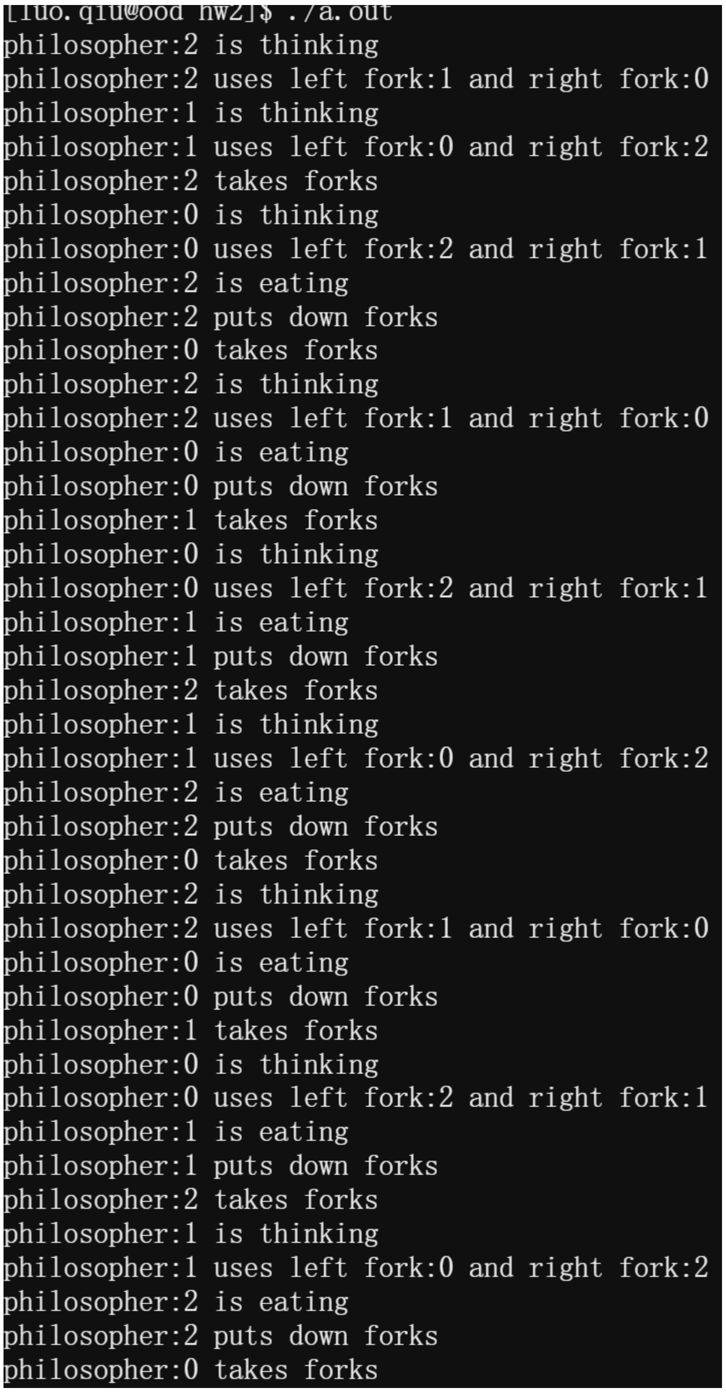
Implementation:
If want to determine the number of philosophers and forks, just set the P_NUMBER through #define in the source file Question1.cpp
The philosopher have 4 states: thinking, take forks, eating and put down forks, and my deadlock preventing strategy is that only allow philosophers to eat when there are two avaliable forks next to him.

Results (5 philosophers):



```
[luo.qiu@ood hw2]$ ./a.out
philosopher:0 is thinking
philosopher:0 uses left fork:4 and right fork:1
philosopher:0 takes forks
philosopher:3 is thinking
philosopher:3 uses left fork:2 and right fork:4
philosopher:4 is thinking
philosopher:4 uses left fork:3 and right fork:0
philosopher:4 takes forks
philosopher:2 is thinking
philosopher:2 uses left fork:1 and right fork:3
philosopher:1 is thinking
philosopher:1 uses left fork:0 and right fork:2
philosopher:0 is eating
philosopher:0 puts down forks
philosopher:4 is eating
philosopher:4 puts down forks
philosopher:3 takes forks
philosopher:2 takes forks
philosopher:0 is thinking
philosopher:0 uses left fork:4 and right fork:1
philosopher:4 is thinking
philosopher:4 uses left fork:3 and right fork:0
philosopher:3 is eating
philosopher:3 puts down forks
philosopher:2 is eating
philosopher:2 puts down forks
philosopher:0 takes forks
philosopher:1 takes forks
philosopher:3 is thinking
philosopher:3 uses left fork:2 and right fork:4
philosopher:2 is thinking
philosopher:2 uses left fork:1 and right fork:3
philosopher:0 is eating
philosopher:0 puts down forks
philosopher:1 is eating
philosopher:1 puts down forks
philosopher:4 takes forks
philosopher:3 takes forks
philosopher:0 is thinking
philosopher:0 uses left fork:4 and right fork:1
```

Results (3 philosophers):



```
[luo.qiu@ood hw2]$ ./a.out
philosopher:2 is thinking
philosopher:2 uses left fork:1 and right fork:0
philosopher:1 is thinking
philosopher:1 uses left fork:0 and right fork:2
philosopher:2 takes forks
philosopher:0 is thinking
philosopher:0 uses left fork:2 and right fork:1
philosopher:2 is eating
philosopher:2 puts down forks
philosopher:0 takes forks
philosopher:2 is thinking
philosopher:2 uses left fork:1 and right fork:0
philosopher:0 is eating
philosopher:0 puts down forks
philosopher:1 takes forks
philosopher:0 is thinking
philosopher:0 uses left fork:2 and right fork:1
philosopher:1 is eating
philosopher:1 puts down forks
philosopher:2 takes forks
philosopher:1 is thinking
philosopher:1 uses left fork:0 and right fork:2
philosopher:2 is eating
philosopher:2 puts down forks
philosopher:0 takes forks
philosopher:2 is thinking
philosopher:2 uses left fork:1 and right fork:0
philosopher:0 is eating
philosopher:0 puts down forks
philosopher:1 takes forks
philosopher:0 is thinking
philosopher:0 uses left fork:2 and right fork:1
philosopher:1 is eating
philosopher:1 puts down forks
philosopher:2 takes forks
philosopher:1 is thinking
philosopher:1 uses left fork:0 and right fork:2
philosopher:2 is eating
philosopher:2 puts down forks
philosopher:0 takes forks
```

a) Does the number of philosophers impact your solution in any way? How about if only 3 forks are placed in the center of the table, but each philosopher still needs to acquire 2 forks to eat?

   The number of philosophers didn’t impact my solution because I take measures to prevent deadlocks.

b) What happens to your solution if we give one philosopher higher priority over the other philosophers?

Giving one philosopher higher priority over the other philosophers is another strategy to prevent deadlocks and that means at every time, at least one philosopher is always able to eat. Since my solution has a different deadlock strategy, this action does not affect the normal operation of the program, and the philosopher whose priority is higher will eat more often.

c) What happens to your solution if the philosophers change which fork is acquired first (i.e., the fork on the left or the right) on each pair of requests?

Changing the order of forks doesn’t impact my solution since the deadlock measure I take is to only allow philosophers to eat when there are two avaliable forks next to him.