The dining in hell problem

The dining in hell problem is quite interesting, for me it had two different interpretations with two different solutions.

The fist one is when the person being fed has to know about it, it can't just accept any random spoon, there has to be some communication.

The solution to this interpretation pairs the threads together and makes them feed each other. Fairness is achieved by having a thread that is waiting to be fed be the first one to be paired, and using the fact that ReentrantLock can be set to a fair mode that gives locks to threads that have waited the longest.

This implementation is really convoluted in the pairing, it has multiple locks, waiting on condition variables and implicit lock passing even though lock passing is not allowed in java. But the big picture looks like this: Have a thread arrive in a monitor, if it's the first thread, it waits until another comes along, releasing the lock.

When another thread comes it acquires the lock and doesn't let it go until the communication with the waiting thread finishes. The waiting thread is then woken up using another lock and a condition variable and using this lock and condition variable, the threads communicate to ensure that both finished and exit the method. On exit the method return the number of the thread we are supposed to feed, and it the calling thread will have the ownership of a lock inside of the monitor object.

After feeding, another method has to be invoked to release the lock, and before going to pair itself again, the thread waits until it's feeding lock is free to not block other threads from pairing up.

Another interpretation and solution (Better)

The other solution assumes that a thread can be just fed, without communication, it just has to be insured that there is nobody else feeding.

This makes the implementation cleaner, as we just have to keep a queue with threads to feed and when somebody calls to feed, return the number and lock of the thread that he will feed.

Implementing a fifo queue for just five philosophers is impractical, so I used a counter that loops around. The idea is that the counter will point to the thread that is to be fed next, and if it's already beng fed then just skip. That way there will be no starvation, as all threads have to be fed in order.

The method feed, does just that, it also throws an exception if we are already holding any lock, as we cannot feed two threads at once, and doesn't allow the thread to feed itself.

This implementation also features the end_feeding method, that releases the lock.

This is the better implementation and is featured in the file hell2.java.

Also the state of the program doesn't print a trace, but refreshes the screen after each update

A sample execution of the solution

```
mint@mint-Legion-7-16ACHg6:~/Documents/concurrent_programing/HM4$ java Main 5
I am 0 and I'm feeding 1
I am 4 and I'm feeding 2
I am 3 and I'm feeding 4
I am 1 and I'm feeding 0
I am 2 and I'm feeding 3
I am 0 and I'm NOT feeding
I am 4 and I'm NOT feeding
I am 2 and I'm NOT feeding
I am 0 and I'm feeding 3
I am 0 and I'm NOT feeding
I am 1 and I'm NOT feeding
I am 2 and I'm feeding 0
I am 3 and I'm NOT feeding
I am 4 and I'm feeding 1
I am 1 and I'm feeding 2
I am 0 and I'm feeding 3
I am 4 and I'm NOT feeding
I am 4 and I'm feeding 1
I am 2 and I'm NOT feeding
I am 0 and I'm NOT feeding
I am 1 and I'm NOT feeding
I am 3 and I'm feeding 4
I am 2 and I'm feeding 0
I am 3 and I'm NOT feeding
I am 4 and I'm NOT feeding
I am 4 and I'm feeding 1
I am 4 and I'm NOT feeding
I am 0 and I'm feeding 2
I am 3 and I'm feeding 4
I am 0 and I'm NOT feeding
I am 2 and I'm NOT feeding
```