16-124-836
Marcel ZAUDER

1 Savages

Because most of the code is used in both scenarios of the exercise, there are many abstract classes and interfaces such that the code can be reused and is not duplicated. For understanding the used components are described in the following section:

- **ICook/Cook:**
  A runnable that can refill the FoodPot when it is empty. Also a method is implemented to fetch the state whether the cook is preparing the meal or not - so that only one savage is ordering the refillment. The *run()* method consists of a while loop which calls the cookAndRefill method after a savage has ordered. Furthermore a terminate work method is implemented such that the CookThread can be "killed" if necessary.
- **IPot/FoodPot:**
  A FoodPot that can be refilled if empty and if not empty a savage can get a portion of food from it.
- **ISavage/OnceHungrySavage/AlwaysHungrySavage:**
  This runnable is able to eat and to order a refill which will lead the cook to prepare and refill the meal.
- **IExecutor/Ex1Savage1/Ex1Savage2:**
  This class contains the main method for executing the simulation. The methos *isAlwaysHungry()* is needed for the two different scenarios explained in the exercise. The method *getLock()* returns the lock which is in use. For the different scenarios different lock will be in use.

The *Cook* and *Pot* are exactly the same for both scenarios and therefore only one implementation of them is needed. Because the Savage class is responsible for fairness we need different implementations for the two scenarios. Also the Executor differs in detail the *isAlwaysHungry()* property and the type of *Lock*.
The *AbstractSavage#run()* first will lock a lock. Then he checks whether the FoodPot is empty and if the cook is not already preparing. If this is the case he will order for a refill such that the Cook can *prepareAndRefill()*. If the FoodPot is not empty anymore he will get one portion of food and afterwards call the *initWaitForTurn()* method and releases the *lock*. The *waitForTurn()* method is responsible for implementing fairness which will be explained in the section 2.2. The *lock* is required such that only one savage can interact with the FoodPot such that a misbehaviour of the pot and a false state of it can be prevented.

1.1 Ex1Savages1

The implementation of the unfair scenario uses a normal *ReentrantLock* to ensure that only one savage can interact with the FoodPot at a time. Furthermore as no fairness is required to be implemented the *OnceHungrySavage* does not need any specification in the *initWaitForTurn()* and *waitForTurn* methods which can therefore be empty.

1.2 Ex1Savages2

The goal for this implementation was to additionally add the constraint of NO STARVATION fairness so that if the savages are always hungry everyone of them can eat at some time and not starve. This property was implemented by each Savage is aware of how many accesses there were since the last time he has eaten. The Savage will only try to acquire the lock if every other Savage has already eaten since the last time he interacted with the pot. To achieve this the *ReentrentLock* is modified such that the accesses can be observed by using a Wrapper class called ObservableLock. This *lock* is then able to notify all its subscribers (happens when the lock is released) so that each savage can count the number of accesses and therefore can define a countdown to determine when his next turn will be. Because of this type of communication between the savages a lock around the critical section might not be needed after the first eat procedure because after that every savage will wait for his turn to interact with the pot.

2 Dining Philosophers

The Executor class is responsible for creating and handling all threads and initializing the components. Each *DiningPhilosopher* and *PhilosopherFork* are put into arrays which therefore represent seating arrangement. When initializing a *DiningPhilosopher* two in the array adjacent forks are used as parameters which can be seen as the philosopher's left and right fork. Because we always increase the iterator i we can assure that no two philosophers have the same two forks "adjacent" to them.
A fork is a modified lock which state of being locked can be easily accessed with the *isTaken()* method.
A *DiningPhilosopher* will try to acquire both forks by locking the forks the philosopher was initialized with. If it was not possible to lock both, whether one is already taken or something else happened, the philosopher will release the fork which he might have already acquired and then try again to lock both. This ensures that this procedure does not lead into a deadlock where each philosopher has one fork and waits for the other fork to be released. Furthermore it ensure a state of fairness because every philosopher will eventually have the chance to acquire both forks. After the philosopher has taken both forks he will eat and the release the forks. In the end he will sleep for a random time between 1 and 1000 milliseconds.
Currently it is fixed on a distinct amount of iterations so the project will eventually terminate but it would be also possible to make it infinitely looping.