Use of Semaphores

Homework 3

Alex Lindberg
alex5@kth.se

February 18, 2022

Introduction

This is a short report detailing the third homework — *Use of Semaphores*. Implementations for task 1, 2 and 3(b,1) are included. All implementations were done in Java using only java `Thread`s and `Semaphore`s.

1 The Hungry Birds

The implementation uses two sub-classes, `ParentBird` and `BabyBird`, which share a resource `worms`. The sub-classes are runnable, meaning we just need to provide an instance of these classes to a thread which will run it for us. The solution to this task was based on using two semaphores acting as mutexs. These are initialized to 1 and 0 permits respectively. The first one is for ensuring only a single baby can eat while the second semaphore is for ensuring the parent can refill the pool of worms.

The implemented solution should be fair. The consumers (baby birds) can't consume food if there is none, and the parent won't refill if there is. The consumer that finishes the food supply signals the producer to refill, and when the producer is finished it releases and waits for the consumers to finish.

```
\L3> java birds.Nest 5
Bird 1> *slurp* nom (5)
Bird 2> *slurp* nom (4)
Bird 4> *slurp* nom (3)
Bird 3> *slurp* nom (2)
Bird 5> *slurp* nom (1)
Bird 1: !!!*CHIRP*!!!
Bird 3: !!!*CHIRP*!!!
Parent> Refilled with 5 worms
Bird 4> *slurp* nom (5)
```

2 The Bear and Honeybees

Just like in Hungry Birds this solution uses two sub-classes that interact with each other using two semaphores. The semaphores are set up in the opposite way from the other solution, i.e. the "consumer" semaphore waits for the "producer" semaphore to finish.

One thing implemented here that ensures that the consumers semaphore is resets correctly is the use of the `drainPermits` function, which simply resets the amount of semaphores to 0.

This solution, similarly to Hungry Birds, is fair. The consumer (the bear) will not consume honey if the pot isn't full, and the producers (bees) will signal the bear if it is. The consumer that finishes the food supply signals the producer to refill, and when the producer is finished it releases and waits for the consumers to signal it again.

```
$ /L3> java bear.Pot 5 5
Bear> *snooooree*
Bee 1> ~Bzz~ *spits honey*
Bee 2> ~Bzz~ *spits honey*
Bee 5> ~Bzz~ *spits honey*
Bee 3> ~Bzz~ *spits honey*
Bee 4> ~Bzz~ *spits honey*
Bee 4> ~BzzZzZZZZz?!~
Bear> *yawn* Snacc time :)
Bear> *yawn* sleepy
Bee 3> ~Bzz~ *spits honey*
```

3 The (fair) One-Lane Bridge

This solution is for problem **3 + (b)**, choosing modification 1. Taking great inspiration from Lecure 10 on semaphores and fairness in the readers/writers problem this implementation uses baton-passing to ensure fairness. We delay a new reader when a writer is waiting, vice versa when readers are waiting. If there are waiting writers we awaken one when a reader finishes and likewise but the other way around when writers finish.

```
$ /L3> java bridge.Bridge 4 2
Car 1> Crossing South with 2 trips left
Car 3> Crossing South with 2 trips left
Car 2> Crossing North with 2 trips left
Car 3> Crossing North with 1 trips left
Car 1> Crossing North with 1 trips left
Car 4> Crossing North with 2 trips left
Car 3> Crossing South with 1 trips left
Car 2> Crossing South with 1 trips left
Car 1> Crossing South with 1 trips left
Car 4> Crossing South with 1 trips left
Car 3> Crossing North with 0 trips left
Car 4> Crossing North with 1 trips left
Car 1> Crossing North with 0 trips left
Car 2> Crossing North with 1 trips left
Car 2> Crossing South with 0 trips left
Car 4> Crossing South with 0 trips left
```