COSI 131a: Fall 2014

Programming Assignment 2

Due date: Part 1: October 27, 2014 11:55 PM ET

Part 2: November 3, 2014 11:55 PM ET

Overview

Your task is to implement a simulation of anthills and ant-eating animals using Java Threads. In this simulation there will be few anthills, but many animals. There are several rules that govern when an animal may eat at an anthill. Your program is responsible for enforcing these rules at all times.

The project is split into 2 required tasks and 1 extra-credit task. Task 1 requires you to use the synchronized keyword and busy-waiting to prevent race conditions. Task 2 requires you to build on your synchronized code and add monitors to avoid busy-waiting and add a scheduler. The extra credit task asks you to use monitors to make the scheduler preemptive.

Task 1 has an earlier due date than Task 2 and the extra-credit assignment. The extra-credit assignment will be released a week after this assignment. . See LATTE for due dates. Your code must be compilable using Java 1.7 - do not use any Java utilities only available in version 1.8

Expected Behaviors

The following rules limit when, where and with whom animals may eat. Your code must enforce these rules in all Tasks.

- Animals (General)
    - Have the following attributes: name, type, colour, hunger, speed, priority
    - Type refers to the kind of animal - Aardvark, Anteater, or Armadillo.
    - Don't like being around animals of the same colour so all animals at the same anthill at the same time must be of a different colour.
    - Hunger limits how many ants an animal will eat total.
    - Speed that determines how many milliseconds it takes the animal to eat at an anthill after successfully gaining access to said anthill.
    - Every animal has a name and a toString() method. The toString() method must return “COLOR TYPE NAME” i.e. “GREEN ANTEATER AA1”.

    - Animals have priority based on their type
        - Priority is a number between 0 and 4 (inclusive).
        - A higher number means higher priority.
- Aardvark
    - Only two aardvarks may eat at an anthill at any given time.
    - Aardvarks will not eat at the same anthill as an anteater at the same time
    - All aardvarks have priority 1
    - All aardvarks have speed 8
    - All aardvarks start with hunger level 3
- Anteater
    - Only one anteater may eat at an anthill at any given time.
    - Anteaters will not eat at the same anthill as an aardvark at the same time.
    - All anteaters have priority 2
    - All anteaters have speed 4
    - All anteaters start with hunger 3

- Armadillos
    - Armadillos will never start eating from an anthill alone.
    - Armadillos will never eat from an anthill with another armadillo at the same time.
    - Armadillos will eat from an anthill with an aardvark (or two aardvarks) or anteater at it.
    - All armadillos have priority 2
    - All armadillos have speed 6
    - All armadillos start with hunger 2
- Anthills
    - Have a limited number of ants and an animal cannot try to eat an ant from an empty anthill

As you know from learning about concurrency, without proper synchronization, these constraints could be violated since a thread may secure permission to eat at an anthill, then be interrupted before it can actually go to the anthill, at which point another thread may also secure permission to eat at the anthill and do so. When the first thread to get permission to eat at the anthill does so, one of the invariants may be violated (e.g., maybe an anteater and aardvark will be eating at the same anthill at the same time).

Part of your job in this assignment is to use synchronization to prevent such race conditions from occurring regardless of scheduling.

Detailed Terminology

- Hunger - determines how many anthills an animal eats before stopping. Each time an animal successfully visits an anthill their hunger must decrement. Once an animal has reached hunger 0, it stops and will no longer visit anthills.

- Speed - determines how long it takes an animal to eat at an anthill. A higher speed means an animal waits at an anthill for less time. The exact formula for waiting time is *((9 - speed) * 100) ms*. I.e. an aardvark with speed 8 will gain access to an anthill, wait for 100ms then leave the anthill.
- Priority - determines order when waiting for an anthill, this is used in part 2 only. If multiple animals are waiting for an anthill to become available, the animal with the highest priority *that does not violate the other constraints* will gain access to the anthill first.

Provided Code

The code for this assignment is divided into the following packages

edu.brandeis.cs131.Ants.AbstractAnts

The AbstractAnts contains the abstract classes and interfaces that your program must implement. **You are not allowed to modify any of the files within this package.**

- Animal.java
- AntFactory.java
- Anthill.java
- Colour.java

edu.brandeis.cs131.Ants.AbstractAnts.Log

The AbstractAnts.Log package contains classes used to record the actions of animals. These classes are used by the test packages only to evaluate that constraints have not been violated. **The log is not a means for passing messages.** The only reference to Log classes in your code should be in the ConcreteAntFactory.createNewScheduledAnthill method. **You are not allowed to modify any of the files within this package.**

- AntEvent.java
- AntEventType.java
- AntLog.java
- DummyLog.java

edu.brandeis.cs131.Ants.YourName

The *YourName* package contains a few mostly empty classes that implement the classes found in AbstractAnts. You should initially rename this package to your first then last name with no spaces i.e. JohnSmith. All of the code pertaining to your solution of this assignment must go in this package. You are free to add, edit, rename, and delete files in this package in the course of implementing your solutions.

- Aardvark.java
- Anteater.java

- Armadillo.java
- BasicAnthill.java
- ConcreteAntFactory.java
- MyAnimal.java

Test Packages - test/edu.brandeis.cs131.Ants

You must modify AntFactoryProxy to include your version of ConcreteAntFactory (Modify the import statement to include your ConcreteAntFactory), in order for the test suite to run. You may not edit any of the other test classes, but we highly encourage you to study them for your understanding and to aid discovering why a test is failing.

- AntFactoryProxy.java
- AntTest.java
    - Contains no test but some methods common to other test classes
- BehaviorTest.java
    - Contains tests that evaluate whether the expected behaviors are adhered to in a single threaded environment by simulating a specific schedule.
    - All test should pass once part 1 has been completed.
- SchedulerTest.java
    - Contains tests that evaluate whether scheduled anthills perform as described.
    - All test should pass once part 2 has been completed.
- SimulationAnthillTest.java
    - Contains two test one using basic anthills only, and one using scheduled anthills. Test simulate 215 animals eating from 10 anthills, and the log is validated to ensure no constraint violation occurred within the test.
    - Basic_Anthill_Test should pass when part 1 is completed.
    - Scheduled_Anthill_Test should pass when part 2 is completed.

API Docs

You will need to understand how all provided code works to solve this problem, even though you will not be allowed to modify it in tasks 1 and 2. You will want to inspect the java files yourself, but to get you started here are the API docs generated with javadoc from the files provided with PA2.

Important Note about Disallowed Java Tools

In PA1, you were instructed to consider using a synchronized class provided by Java for inter-thread communication (ArrayBlockingQueue) to solve the problem. For this project, that is **not allowed**; you may not use **any** synchronized data structure included in the Java API. You must write your own (using the "synchronized" keyword). Of course, you can and should use non-synchronized data structures in the Java standard library. You can consult the API docs to see if a data structure is synchronized.

You also **may not** use the thread priority methods provided by Java (e.g., you may not use Thread.setPriority).

Task 1: The Basic Anthill

Implementation

You must create a class (a shell file called BasicAnthill has been provided) that implements the interface Anthill and carries out the following tasks:

- Enforce the Anthill Restrictions described previously.
- Use the Java synchronized keyword to prevent race conditions (without introducing deadlock).

This class must be generated when createBasicAnthill is called from your AnthillFactory class. You must define the methods for generating the individual animal classes within your AnthillFactory. Each animal must behave as defined in the animal rules and behavior section, except you will not handle priority scheduling within this task.

Task 2: The Priority Scheduler

You have now successfully programmed BasicAnthill which enforces entry criteria and prevents race conditions. However, there are two problems with the design of BasicAnthill:

- While no anthill is available to an Animal, that Animal busy-waits (loops over all anthills repeatedly inside of its run() method).
- There is no way of prioritizing animals, all animals priority are currently ignored.

In this task you will solve the same problem but by using Java monitor synchronization to avoid busy waiting.

Implementation

You must create a new class (ScheduledAnthill) that implements Anthill and controls access to a collection of Anthills in order to implement the priority scheduling policy defined in the Behavior section. The Anthills "behind" the ScheduledAnthill will be BasicAnthills. BasicAnthill should be the same class you implemented for the first part of this project. The ScheduledAnthill must be returned by the AnthillFactory's createNewScheduledAnthill method.
ScheduledAnthill will carry out the following tasks:

- Keep references to BasicAnthills as private member variables inside ScheduledAnthill.
- When an Animal calls tryToEatAt(animal) on a ScheduledAnthill instance, the following algorithm should be run:

  If animal.getPriority() < the highest priority from among all the other animals waiting to eat at an anthill (i.e., there is another animal with higher priority),

or there are no other animals with higher priority but there are no anthills which the animal can currently eat at,
then the animal thread must wait;
otherwise the animal successfully starts eating at one of the anthills.

- When an Animal exits the scheduler by calling exitAnthill on a ScheduledAnthill instance, the scheduler must call exitAnthill(a) on the appropriate anthill from the collection of anthills managed by the scheduler (remember, you may **not** modify Animal or any of the other classes provided to you, or BasicAnthill, so you must solve this problem within ScheduledAnthill).
- Use monitors to avoid busy-waiting if an animal cannot find an anthill to eat at.
- Maximize concurrency by synchronizing only where necessary.

Hints

instanceof operator

You might find the instanceof operator (scroll down a bit on the page that links to) helpful. instanceof can tell you if an object can be downcast from a parent type into a child type. Typically, we don't use it because there are better techniques (polymorphism leverages the type system to do the work for you), but for this problem it will probably help you out.
Here is an example:

```
public class TestInstanceof {
    public static class Parent {
    }
    public static class Child1 extends Parent {
    }
    public static class Child2 extends Parent {
    }

    public static void main(String[] args) {
        Parent p1 = new Child1();
        Parent p2 = new Child2();

        if(p1 instanceof Child1) {
            System.out.println("p1 is instance of Child1");
        }
        if(p1 instanceof Child2) {
            System.out.println("p1 is instance of Child2");
        }
        if(p2 instanceof Child1) {
            System.out.println("p2 is instance of Child1");
        }
        if(p2 instanceof Child2) {
```

```
            System.out.println("p2 is instance of Child2");
        }
    }
}
```
This outputs:
p1 is instance of Child1
p2 is instance of Child2

synchronized blocks

The following two uses of the synchronized keyword are equivalent:

```
public synchronized void m() {
    // ...
}
public void m() {
    synchronized(this) {
        // ...
    }
}
```

If you want to synchronize the whole method on the lock contained inside of "this", then it is more elegant to put the synchronized keyword in the signature of the method.
Sometimes, you want to synchronize on smaller blocks of code than an entire method. This allows you to limit the amount of synchronization, which can improve performance by **maximizing concurrency**. In this case, putting synchronized in its own block can be the better choice.

```
public void m() {
    // a non-critical section
    synchronized(this) {
        // a critical section
    }
    // a non-critical section
    synchronized(this) {
        // a critical section
    }
    // a non-critical section
}
```

Note that you can synchronize on the lock in any instance, not just "this". For example:

```
public class C {
    private static Object o = new Object();

    public void m() {
        synchronized(o) {
            // ...
        }
    }
}
```

You might not need to synchronize on an instance other than "this" for this project.

Submission Guidelines

Create a zip or tar.gz file containing your completed code. You should name this file: last name underscore first name. For example if your name is Amanda Bynes you should name your file: Bynes_Amanda.zip or Bynes_Amanda.tar.gz. This file should contain the src folder within the archive -not as the top level (when the archive is opened the src folder should be visible). Upload this archive to the submission folder on Latte. Remember your code should be compilable using Java 1.7. Please also include a README file that contains a description of your implementation and any misbehavings your code might have. This is an individual project and all submitted work must be your own, however you are allowed to discuss your solution with other students, **do not share your code**. If you have extensive discussions with another student please indicate this in your README file.