JavaPathFinder Exercises

<u>**27th Nov 2007**</u>

Preamble

In each exercise, you will be required to write Java programs based on some starter files, and run them through JPF. You will have to report your final Java files, and the *time* and *memory* required by JPF to verify these files. You must report your time and memory measurements in the following tabular format:

Machine Spec	CPU = XYZ GHz PQR Cores	Memory = XYZ MB
Key	*Time (seconds)*	*Memory (MB)*
N1	T1	M1
N2	T2	M2
N3	T3	M3
…	…	…

The *Key* entry will be specified in each exercise. You must increase the value of *Key* till your experiment times out at 900 seconds (you must indicate this by a "*" in the *Time*) or your machine starts thrashing due to lack of memory (in which case you must put a "*" under *Memory*).

Exercise 1: Dining Philosophers

In the first lecture we saw a version of the Dining Philosophers (DP) problem that had a deadlock. There are a number of ways to avoid deadlock with DP. In this exercise, we will model and verify two such approaches. The generic command line to run any of the following DP programs through JPF is:

```
jpf <DP class name> <number of philosophers>
```

Exercise 1.1. Dining Philosophers with Deadlock

The starter file *DP.java* contains a version of DP with deadlock. Run this program through JPF and make sure that JPF does indeed find the deadlock. Tabulate your results as shown above using the number of philosophers as *Key*.

Exercise 1.2. Using one Left-Handed Philosopher

We will remove the deadlock by forcing one of the philosophers to pick up her forks in a different order than the remaining philosophers. The file *DPDifferentOrder.java* has the basic framework to do this. Right now it is exactly the same as *DP.java.* You must modify it appropriately to achieve the desired result. You must not add any more methods, fields or classes, but simply modify existing methods. Report your Java file and experimental results using the number of philosophers as *Key*.

Exercise 1.3. Using a Butler

We will remove deadlock by using a butler that allows at most (*K – 1*) philosophers from being at the table, where *K* is the total number of philosophers. The starter file *DPButler.java* contains a basic infrastructure for using a butler. However it has an error and will deadlock. Fix the error and run JPF to ensure that deadlock is indeed no longer possible. You should do this without adding a new classes, fields or methods. Report your Java file and experimental results using the number of philosophers as *Key*.

Exercise 2: Santa Claus

A simple version of the Santa Claus problem is defined as follows: Santa has *M* reindeer and *N* elves[1]. Santa repeatedly sleeps until wakened by either: (a) one of his *M* reindeer, back from its holidays, or (b) by one of his *N* elves. If awakened by a reindeer, he harnesses it to his sleigh, delivers toys with it and finally unharnesses it (allowing it to go off on holiday). If awakened by an elf, he shows him into his study, consults with him on toy R&D and finally shows him out (allowing him to go back to work).

The file *Santa.java* contains a partial implementation of the Santa Claus problem. Run the program through JPF using the following command:

```
jpf +vm.por.sync_detection=false Santa <# of reindeer> <# of elves>
```

Our main goal is to verify the following safety property: **(PROP)** whenever Santa is woken up, there is exactly one reindeer or one elf waiting for him. We will do this in three stages.

Exercise 2.1. Using Assertions

The program supplied to you already contains the right variables to check for **PROP**. However, these variables are not being set properly, nor is the appropriate condition over these variables being asserted. Your task is to: (a) identify the right variables (Hint: they are in the *SantaClaus* class), (b) set these variables properly, (c) add an appropriate assertion over these variables that checks for **PROP**, and (d) verify that these assertions are never violated using JPF. Report your Java file (rename it to *Santa1.java*) and your results in a tabular format using (*M,N*) as your key. Start with *M = 2* and *N = 2,* and slowly increase their values till your computer times our or runs out of memory. Increase the value of *M* and *N* alternately, starting with *M.* In other words, your key values should be (2,2), (3,2), (3,3), (4,3), (4,4), (5,4), (5,5) and so on.

Exercise 2.2. Using Property

Verify **PROP** by implementing a new class that extends *GenericProperty* as described in the first lecture. Specifically, you must do the following: (a) identify the right variables, (b) set these variables properly, (c) implement a class called *SantaProperty* that extends *GenericProperty* and checks for **PROP**, and (d) verify that **PROP** is never violated using JPF, i.e., verify that the *check()* method of *SantaProperty* never returns *false*. Report your Java file (rename it to *Santa2.java*) and your results as in Exercise 2.1.

Exercise 2.3. Using Listeners

Verify **PROP** by implementing a new class that extends *PropertyListenerAdapter* as described in the first lecture. Specifically, you must do the following: (a) implement a class called *SantaListener* that extends *PropertyListenerAdapter* and checks for **PROP**, and (c) verify that **PROP** is never violated using JPF, i.e., verify that the *check()* method of *PropertyListenerAdapter* never returns *false*. Report your Java file (rename it to *Santa3.java*) and results as in Exercise 2.1.

Exercise 3: Standard Santa Claus (Extra Credit)

The Santa Claus problem is defined as follows: Santa has *M* reindeer and *N* elves. Santa repeatedly sleeps until wakened by either: (a) all *M* of his reindeer, back from their holidays, or (b) by three of his *N* elves. If awakened by the reindeer, he harnesses them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by three elves, he shows them into his study, consults with them on toy R&D and finally shows them out (allowing them to go back to work). Your job is to implement the Santa Claus problem in Java using separate threads for Santa, the reindeer and the

[1] This is a slightly modified version of the standard Santa Claus problem.

eleves. Your implementation should satisfy the following safety property: **(PROP)** whenever Santa is woken up, he works with either exactly M reindeer or three elves. Verify this property on your program using the three techniques described in Exercise 2.