Lab Session 2, Concurrency and Parallelism

Jean-Viven Millo
INRIA Sophia-Antipolis
jean-vivien.millo@inria.fr

Andrea Tettamanzi
UNSA
andrea.tettamanzi@unice.fr

March 10, 2014

Abstract

The goal of this session is to discover the basic mechanisms of semaphores and priorities.

Then we will move to the communication through sockets in JAVA. An Eclipse project containing the skeletons of the source files is available at the same address as this document. In order to realize the exercises, you will require the following documents: i) this lab session subject ii) the skeleton of the source files iii) the course iv) the JAVADOC (http://docs.oracle.com/javase/1.4.2/docs/api/).

Exercise 1: Semaphore

Taken from *http://en.wikipedia.org/wiki/Dining_philosophers_problem*
Five silent philosophers sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it's not being used by another philosopher. After he finishes eating, he needs to put down both the forks so they become available to others. A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that each philosopher won't starve, i.e. can forever continue to alternate between eating and thinking, assuming that any philosopher can not know when others may want to eat or think.
1/ Define the *fork* to be a class with only a boolean state: "available" or "currently used" and the associated getters and setters.
2/ Implements the dining philosophers problem in a way that avoid deadlock.
3/ Have you *synchronized* the accesses to the forks?
4/ Change the implementation in order to use the class **java.util.concurrent.Semaphore**.

Exercise 2: Readers writers with priority

Let us consider a table of 1000 bytes that will simulate the behavior of a memory. The access to this memory are the following:

1. There are three writers that are reading data from text files and writing them in memory at he next available slot.
2. There are three readers that read the next available data in the memory and write them in (another) text files.
3. The writers have priority over the readers. Of course, when the memory is full, writers cannot write anymore.
4. The three readers as well as the three writers have there own hierarchy of priorities.
5. The three readers as well as the three writers alternate between sleeping for a random time and writing 100 bytes in the memory.

1/ Implement the system as described
2/ Let the readers and writers work on a random size (≤ 1000) packet of data instead of 100.
3/ Consider the following extra requirement: Reader i reads only data from Writer i (with $i \in \{1, 2, 3\}$).

Exercise 3: Client/Server

1/ Write a class *Server* that:

- opens a socket on a port number given as an argument,
- waits for a connection,

- prints a message when the connection is accepted.

2/ Try to connect to the server using i/ the *Telnet* command and ii/ using a browser

3/ Write a class *Client* that connects to the server and prints a message when it is done. The address and port of the server are given as an argument.

4/ Improve your application so that the client sends successively 5 messages and receives an acknowledgment including the original message from the server. When the client emits the keyword **stop**, the connection are stopped at both end and the program is shut down.

Exercise 4: Multi-client

Modify the server part of the application written in Exercise 1 so that the server can accept many client connections.

1/When the server accepts a connection, it delegates the management of this connection to an instance of the runnable class *ClientManager*, and then wait again for connection.

2/The class *ClientManager* discusses with the associated client similarly to the server in Exercise 1.