IN325: 1 – Concurrency in Java

Author : Christophe Garion <garion@isae.fr>
Audience : 3A IN
Date :

Abstract

The objectives of this lab session is to manipulate the basic concurrency API in Java.

1 Multiplication of matrices

We consider the multiplication of two matrices A and B of size $m \times p$ and $p \times n$. The matrices will contain values of type **`float`** and we supposed that they are filled by 1 (to easily verify the result of your computation...).
We want to use threads to compute the result of $A \times B$: each thread will have a certain number of rows of the results to compute.
Write a Java program such that:

- you specify the m, p and n parameters from the command line;
- you specify the number t of threads from the command line (the last thread has the remainder of m/t row to compute);
- you verify the results of the multiplication.

Write a single-threaded program doing the same multiplication and compare execution times (choose m, p and n accordingly).

2 A bank account

1. develop a `BankAccount` class whose specification is given on figure 1. Verify in method `withdraw` that the withdrawal is authorized and return **`true`** if the withdrawal has been correctly done.

BankAccount
- balance: double

+ BankAccount(balance_: double)
+ getBalance(): double
+ deposit(money: double)
+ withdraw(money: double): boolean

Figure 1: The `BankAccount` class

2. create a `Client` class representing using threads clients executing the following operations on the same account:
   - for the first client, 1000 deposals of 5 euros and 1000 withdrawals of 10 euros;
   - for the second client, 1000 deposals of 10 euros and 1000 withdrawals of 5 euros.
3. create a test program creating an account with initial balance of 1000 euros and launching the two clients. Do not forget the two thread to complete before printing the final balance. Execute several times the program. What happens? Why?
4. "synchronize" now the `BankAccount` class. What happens?

3 Producers and consumers

1. using basic concurrency API of Java, implement a classical producer/consumer FIFO. We suppose here that:
   - the FIFO can only store one type of messages
   - the FIFO has a fixed size

2. we want now to add the following requirements:
   - the messages have an "urgency" attribute to order them in the FIFO
   - when exceeding the FIFO size, three politics are available:
     - DROP_OLDEST
     - DROP_NEWEST
     - RAISE_EXCEPTION

     The politic will be chosen at FIFO creation. Use a factory design pattern to implement your solution.
   - add a timeout on message waiting
3. implement a mechanism to forbid to a thread to send more than *n* messages by time unit.

4 Using collections

1. create a `CollectionClient` class that implements `Runnable` and add 1000 integers to a list. Create programs that use two `CollectionClient` instances on the same list with:
   1. an instance of `ArrayList`
   2. an instance of `Vector`
   3. an synchronized instance of `ArrayList`

   Those programs will wait the completion of the two threads and print the list size. What happens?
2. create a `CollectionClientConcurrentModification` class that implements `Runnable` and provides two possible behaviour given a list:
   - either add 1 to every element of the list and print the new value;
   - either wait 20ms and add a new interger to the list and print "new value added!".

   Create programs that create a list with 1000 elements and use two `CollectionClientConcurrentModification` instances (one for each case) on the same list with:
   1. an instance of `ArrayList`
   2. an instance of `Vector`
   3. an synchronized instance of `ArrayList`

   Those programs will wait the completion of the two threads. What happens?