Concurrent and distributed programming (Exercises)

2011/2012

Dr. Arno Formella

Departamento de Informática
Universidade de Vigo

11/12

Comments

The programming exercises seem to be easy when you read their description, however, you will notice, they are not that easy once you try to solve them.

Starting I

1. Write the “hello world” program in Java.
2. Write a “hello world, thread ... speaking” program using a set of threads (consider taking a close look to the manual pages of `Thread` and `Runnable` and use both possibilities.)
3. Measure how many threads you can start and keep alive.

Starting II

4. Measure how much time a single thread needs to execute a certain task, e.g., writing 100000 times “hello world”, and how much time a set of, let’s say, 1000 threads needs to perform the same task distributing the work among them. Generate a diagram plotting the execution time over the number of running threads.
5. Change the work to be done by something that does not use output operations, and generate the same plot as above.
6. Make sure that your programs terminate smoothly, i.e., all participating threads reach their final “}”.

Starting III

Describe precisely your observations (dependencies of the results on the operating system, system load, work load, kernel number, etc.).

PingPONG I

1. Implement a perfect `pingPONG`. Consider the following details:
   - Experiment with the different trials presented in the class notes. (Observe and take notes of their properties.)
   - Develop a solution with the following properties:
     1. Use three threads (one thread for the main program, that is the referee, and one thread for each player).
     2. The referee starts the game (with a previous message to the screen).
     3. The players write their `ping`s and `PONG`s, respectively.
     4. The referee stops the game after a certain amount of time has elapsed (again writing previously a message to the screen).

PingPONG II

5. The players exchange the ball at most one more time.
6. Both players/threads stop (writing a corresponding message).
7. The referee writes the last message.
8. The program terminates.

- Observe the difference using `notify()` or `notifyAll()` in the synchronization protocol, especially concerning the number of useless wake–ups of threads.

PingPONG III

2. Extend your program to work with as many players as given in the command line (the maximum number you know from the previous exercise). Generate a table with the execution times for different numbers of players but constant number of ball exchanges (including the trivial case of one player writing only `ping`s).
3. What would be a perfect solution? i.e., an implementation where just the next player who is to play is woken up.
4. Implement the game `pingPONG` between two computers.
   - Assume the IP addresses known beforehand.

PingPONG IV

- Duplicate the output on all participating computers, each one using a different prefix, e.g., `referee:`, `player red:`, and `player blue`.

Process planning with priorities I

Design/implement an application with three types of processes/threads exhibiting three different priorities (let's say A, B, and C) while trying to access one resource.

1. How do you implement the control of the scheduler such that all processes have access to the resource as described in the on-going: within the same priority group, the access to the resource follows the ordering in time, and among the different priorities the accesses should be distributed such that within the last k accesses granted at least $a\%$ are for class A, $b\%$ are for class B and the remaining $c\%$ are for the class C? (k, a, b, and c are

Process planning with priorities II

configuration variables of the scheduler, clearly, the percentages count only if there are processes of a certain class available, their sum cannot exceed 100 %). (Hint: a scheduler is able to count.)

2. Argue reasonable that your solution guarantees *finite* waiting times for all processes that try to get access to the resource.

Concurrent data structures I

1. Preparation:
   - Study closely the package `java.util.concurrent`.
   - Study the implementation of a concurrent list `http://trevinca.ei.uvigo.es/~formella/doc/cd06/ConcurrentList.tgz`.
2. Use the concurrent list to implement a hashtable (or hashmap) in the following way:
   - There is an array of fixed size which is indexed by the keys of the objects. Each field of the array holds a concurrent list that stores the objects with the corresponding key.

Concurrent data structures II

- Implement at least the following operations: `insert` (inserts a new object into the table), `lookup` (returns true if the object is found in the table, otherwise false), and `delete` (deletes an object, if found in the table).

3. Implement a use case of the hashtable sufficiently large so you can realize measurements of execution time.
4. Compare your implementation with a direct usage of the `ConcurrentHashMap` of Java according to execution time, memory consumption, and what you find interesting.