Lab: Threads and Synchronization

Preamble

To do the lab, please download and open the NetBeans project template provided on Campus.

Before running a program, read the code and the comments to understand what it is supposed to do. Also, read carefully the API Javadoc whenever you use a class or a method for the first time.

1 Running and Controlling Tasks With Threads

`Thread`s are the basic construct to run several tasks in parallel.

1.1 Basic Operation

The `thread.ThreadBasicsDemonstrator` class demonstrates the basic operation of threads. Execute the program several times and observe the following:

- When starting, the JVM creates an initial thread in order to execute the `main` method of the class. What is the name of this thread?
- Threads can be created in two ways: (1) by subclassing the `Thread` class, (2) by providing a `Runnable` object to the `Thread` constructor. Which method is more flexible?
- When started, threads run in parallel: the messages they print interleave[1]. Note that the precise scheduling of threads cannot be predicted[2].
- The `main` method has little work to do and terminates well before the other two threads. When `main` terminates, the thread that executes it also terminates.

The `Runnable` interface is a functional interface. Create a new thread, named `thread3`, whose code is defined by a lambda expression. `Thread3` execute the same task as `thread2`.

Next, declare `thread1`, `thread2` and `thread3` as *daemon*, using the `Thread.setDaemon` method. Then run the program again. What do you observe? Explain it.

1.2 Waiting for Tasks to Complete

Although `thread1`, `thread2` and `thread3` are daemon threads, we want the `main` thread to wait for their completion, using the `Thread.join` method. Check that your program works as expected.

1.3 Executing Periodical Tasks

Sometimes one needs to perform periodical tasks, for example monitoring tasks. Modify the code of the class `MyRunnable` so that it prints the message “Hello world!” every second, forever, using the `Thread.sleep` method.

[1] This actually depends on the number and speed of the processors on your computer, and your operating system.
[2] This is the reason why you must run a multithreaded program several times before drawing conclusions.

1.4 Stopping Tasks

Sometimes one needs to interrupt a task, for example because it is taking too long to complete. The `Thread.stop()` method must not be used because it is inherently unsafe. Instead, the thread must be notified that it should stop, which the thread will do when the appropriate time comes – typically when the thread is not updating some shared variable in a synchronized section.

Notifying a thread that it should stop is the purpose of the `Thread.interrupt` method. Use this method to stop `thread2` after 5 seconds of execution.

2 Running and Controlling Tasks With Executors

As of Java 5, it is good programming practice to use `Executors` rather than `Threads`. Executors decouple task definition from task execution, and they provide advanced services. Note, however, that `Executors` use `Threads` behind the scene.

2.1 Basic Operation

Check the `Executor` interface and the `Executors` utility class, which provides several pre-defined executors. Using a fixed thread pool, execute the three tasks of exercise 1.1.

Experiment with various pool sizes – 1, 2, 3 and above. What do you observe? Explain it.

Also, observe that the JVM does not terminate. Explain why and solve the problem.

2.2 Waiting for Tasks to Complete

Now modify your code so that the `main` method prints “All tasks completed” when all the tasks are completed. This can be done using two methods. Which method is easier?

2.3 Stopping Tasks

Now have the “Hello world!” task stop after 5 seconds of execution. The other tasks must continue running.

3 Synchronizing Tasks at Low Level

3.1 Motivation: Memory Consistency Errors

Threads are allowed to cache the data they read or write to increase performance. As a result, when a thread writes to data it shares with other threads, the update might not be visible to the other threads.

The `synchronization.MemoryInconsistencyDemonstrator` class demonstrates this problem. Run the program several times. What do your observe? Explain it in details.

Solve the problem using two methods. Which method is the easiest?

3.2 Motivation: Thread Interference – Lost Updates

Thread scheduling is unpredictable. As result, when two threads concurrently update a shared variable, some interleavings of low-level steps can result in one of the updates being lost.

The `synchronization.LostUpdateDemonstrator` class demonstrates this problem. Run the program several times. What do you observe? Explain it in details.

Does declaring the shared variable `volatile` solve the lost updates problem? Why?

Solve the problem using two similar solutions. Which one is best in the particular case addressed here? Which one is best in the general case?

Observe that the execution time increases after solving the problem. Explain why.

3.3 Motivation: Thread Interference – Inconsistent Reads

Thread interference can be observed in a more simple setting than the previous execise: one thread updates a shared variable (which is an object), and the other thread just *reads* it.

The `synchronization.InconsistentStateDemonstrator` class demonstrates this problem. Run the program several times. What do you observe? Explain it.

Next, solve the problem. Note: this entails significant modifications to the `MyObject` class.

4 Concurrency Utilities

4.1 Atomic Variables

Solve again the lost updates problem by using an `AtomicInteger` object. Which method should you prefer?

4.2 Synchronized Collections

The `utilities.CollectionDemonstrator` class features two threads filling-in an `ArrayList` with 100,000 elements. What do you observe when running the progam?

Check the Javadoc of the `ArrayList` class and (i) explain the problem, (ii) solve the problem.