Lecture 07

Object-Oriented Concurrent Programming

Shared Object Communication Synthesis.

Object-Oriented Concurrent Programming, 2019-2020

v2.2, 31-10-2019

Miguel Oliveira e Silva, DETI, Universidade de Aveiro

Contents

1 Concurrency and Inheritance

- Extending object-oriented programming with concurrency raises several problems. One of them is a possible *negative interference* with the inheritance mechanism and *subtype polymorphism*.
- Subtype polymorphism promotes a higher abstract programming level by separating a type utilization (through its ADT) from one or many type implementations.
- That is, we can use an object without knowing its construction type (only that it complies with a given ADT).
- So, if shared objects are involved in subtype polymorphism, it is necessary to impose the following rule:

Shared Objects Subtype Polymorphism: An ADT of a shared object, forces that all its possible implementations (subclasses of the ADT class type) must also be shared objects (or else, a sequential object could be used where a shared one was expected).

- However, Java does not enforce this rule (in compile time or in runtime). So programmers will be required to comply with it (with all the risks involved).
- Hence, it is a good programming practice to make explicit the indication of shared objects in their types (classes). A good approach is to use the prefix `Shared` in class types.
- In this way, variables that reference such types most likely won't do the mistake of passing a sequential object where a shared one was expected.
- This option is also appropriate to separate (when possible) the sequential logic of the object implementation from its synchronization scheme.
- A very interesting point that should be mentioned, is the fact that in an OO approach to concurrency the shared object's ADT does not impose a particular internal synchronization scheme (other then to comply with all the correctness criteria) .

- Moreover, theoretically nothing forbids that during the lifetime of a shared object its synchronization scheme cannot be changed.
- However, if a shared class contains in its public interface references to other shared types, then we must ensure the transitivity of this sharing property.
- In Java, this requirement may impose the inclusion of concurrency synchronization code (for external synchronized) in the "sequential" logical code implementation, breaking the desired separation between the sequential logical implementation and the synchronization implementation.
- As a *rule of thumb*, shared classes with this "problem" should be avoid.

2 Shared Objects: Summary

- The construction of thread-safe programs depends, in its essence, on the appropriate management in the access to shared mutable states.
- In order not to lost the simple (and powerful) view of objects given by OOP, a shared class should be seen and implemented as an *atomic data type* (regardless of its number of fields).
- Hence, the following object rules should be enforced:
  1. *total coverage*: all public methods of a class should be coherently synchronized;
  2. Respect the *semantics of the ADT*. That is, the class contracts are kept (class invariants and public methods preconditions and postconditions).
- It is important to mention again that an atomic data type need not to be implemented with a mutual exclusion scheme (many other possibilities exists).
- In fact, as we have seen, synchronization of shared objects can be done without locking synchronization schemes such as software transactions.
- Since shared objects safety problems are essentially attached to its mutable state, the smaller it is, the easier it is to make it atomic.

- In general, object sharing communication requires three synchronization aspects: internal (intra-object), external (inter-object), and conditional.
- *Internal synchronization* results from the need of object atomicity. It ensures class invariants are meaningful.
- *External synchronization* results from the logic of the shared object client code. It arises directly from the use of *structured instructions* that select code based on a concurrent condition that depends on the object state (conditional and iterative instructions).
- *Conditional synchronization* results from the fact not all operations on objects are always applicable. It arises from concurrent conditions present in *assertions* (contracts) attached to objects.

- The joint implementation of these three synchronization aspects can rise *coexistence problems*.
- The *group mutual exclusion* scheme, together with *condition variables* attached to internal and external synchronization schemes, is a possible practical solution to these problems.

3 Concurrency Library `pt.ua.concurrent`

- In the beginning (2010) this library was developed for three main reasons:
  - To be a Design-by-Contract based library (both in native Java and in JML);
  - To get rid off the unsafe (and very annoying) `InterruptedException` checked exception;
  - For pedagogical support in this (OOCP) course.
- In summary: both to maximize the correction of code and to ease its development.
- Later one, other reasons had justified its relevance:
  - To implement new synchronization modules (see library documentation);

  - To support new language mechanisms for concurrent programming (Concurrent Contract-Java).

- This library is structured in the following groups of modules:
  - Base library;
  - Utility modules;
  - Elementary synchronization modules;
  - Locking modules;
  - Barrier modules;
  - Message passing modules.
- Documentation:
  `http://sweet.ua.pt/mos/pt.ua.concurrent/index.xhtml`