Lecture 04
Object-Oriented Concurrent Programming
Shared objects

Object-Oriented Concurrent Programming, 2019-2020

v2.3, 09-10-2019

Miguel Oliveira e Silva
DETI, Universidade de Aveiro

Object-Oriented

Concurrent Programming

Object-Oriented Concurrent Programming

- A program contains mechanisms to allow the execution of some actions over some data. These are the three forces involved in computation.

execution

(sequential/concurrent)

actions

(algorithms, functions)

data

(variables, objects)
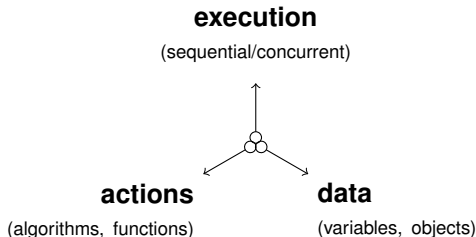
- Procedural programming is structured around the algorithms, while object-oriented programming emphasizes data (though in an abstract way!);
- Concurrent programming aims to take advantage and give full expression to the third dimension: execution.

Object-Oriented Concurrent Programming

- The goal is to join both worlds – objects and concurrency – trying to get the best out of each one and, in the process, control their problems and limitations;
- Reuse the powerful OO techniques for program design and development;
- Maintain the simplicity and expressivity of objects in concurrent contexts;
- Tame the non-determinism inherent to concurrency by using appropriate object implementations that maintain its (explicit or implicit) contracts.

Challenge!

Practical approaches for concurrency that make it simple and safe!

Objects in a concurrent environment

Objects in a concurrent environment

- When concurrency applies to an object-oriented program, two problems arise:
  1. What active concurrent entities (e.g. threads) are allowed to execute within each object?
  2. How can active entities communicate using objects?
- In the first problem, a solution can be prescribed in which the same single active entity is allowed to execute each object.
  - In this case we are in the presence of an actor concurrency model (to be dealt later on the course).
- Another alternative is to allow objects to be shared between active entities (shared object model).

Objects in a concurrent environment (2)

- The solution taken on this first problem leads to a default prescription to the second problem.

- If a single unique active entity is allowed to execute within each object of a program, then communication will naturally through (direct) messages.

- If, on the other hand, objects can be shared, then the indirect shared memory model is the natural solution.

Shared Objects:

Correction

Shared Objects: Correction

- The most important software quality factor to ensure in the introduction of concurrency in OOP is correction.
- In sequential languages the imposition that objects may only be externally used at stable times ensures the simplicity in the application of correction conditions to the object's ADT.
- However, a deregulated concurrent execution context may compromise this simplicity!

Object Concurrent Integrity

Concurrency cannot, in any case, compromise the implementation of the ADT of an object's class.

# Sequential Consistency and Linearizability

Sequential Consistency

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Linearizability

An object is linearizable if each operation appears to take effect instantaneously at some point between the operation invocation and response.

Sequential Consistency and Linearizability (2)

- A sufficient condition to ensure the internal correctness of shared objects (i.e. to ensure its invariant) is the application of mutual exclusion in all of its public operations (monitor)!
- However, the linearizability criteria does not require mutual exclusion.
- In fact, an atomic behavior of shared objects is enough.
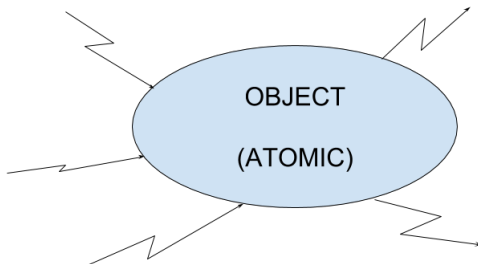
Total Object Covering

Total Object Covering

A necessary condition to safely apply a synchronization scheme to objects, is that all of its public services are consistently synchronized.

Total Object Covering

A correctness condition for the synchronization of concurrent objects is the necessity that all of the object's exported services are protected with an appropriate synchronization mechanism.

(Note: A synchronization mechanism should not be confused with Java's synchronized language mechanism!
A mutual exclusion synchronization scheme is simply one possibility.)

Shared Objects:

Aspects of

Synchronization

Aspects of Synchronization

- internal (intra-object);
  - Ensure the minimum protection of the object's internal state in the presence of a concurrent use
- conditional;
  - Required when the object access depends on a condition that might depend on the execution of other threads
- external (inter-object);
  - To enable the use of one or more shared objects for more than one operation (as if it was a single one)

Internal Synchronization

- Serves to guarantee the basic object correction.
- It should be the responsibility of the object.
- The object should ensure that its invariant is (also) not compromised by unsynchronized uses of its services.
- In the same way, the object should ensure sanity in the execution of all its services.
- A possibility for synchronization that warrants all these conditions is simply impose mutual exclusion in the use of all its public services (we shall see that there are others).

Conditional Synchronization

- Serves to ensure a specific access condition to the object.
- It can be internal, if the condition is directly related with the object internal state and contract;
- Or external, if other condition is externally imposed;
- Requires a process of conditional waiting or the possibility of failure in the access to the object (this failure can be part of try/error cycle until its verification).

External Synchronization

- Serves to ensure a sequential (alike) access to the object for the realization of a sequence of operations;

- The need for this synchronization aspect comes from outside client code, thus failures in this synchronization do not compromise the object's invariant.

- The simplest synchronization scheme for external synchronization is mutual exclusion.

Contracts and Concurrency

Contracts and Concurrency

- The meaning of contracts in a sequential program is simple and clear: an assertion is a correctness condition that must be true in its application point (or else, we are in the presence of a program error.

- It is the type of the false assertion (invariant, precondition, postcondition, etc.) that determines the code responsible for the error.

- What about in a concurrent context?

- What is meaning of a concurrent assertion? (i.e. an assertion with a condition whose truthness may depend on the execution of another thread.)

Contracts and Concurrency

```
assert C end
```

```
if not C then
    throw exception
end
```

```
reserve OBJs do
    if not C then
        throw exception
    end
end
```

```
wait C end
```

RACE CONDITION!

RACE CONDITION!

SAFE!

Contracts and Concurrency

- So we come to the following rule:

Concurrent Assertions

A concurrent assertion only makes sense if it is a waiting condition for the condition!

Contracts and Concurrency

- As a corollary we have that condition synchronization (internal or external) is closely related with assertion semantics.
- It is important to emphasize that an assertion can both be sequential or concurrent depending on its execution context (hence is it a dynamic property).
- That is, an assertion using one or more shared objects might be sequential or concurrent depending of an exclusive access to those objects.

```
assert (CONDITION);
```

```
assert (!lockIsMine() || CONDITION);
if (!lockIsMine()) {
    // waiting cycle for the condition...
}
```

Contracts and Concurrency

- Another very important aspect in the behavior of concurrent assertions is related with the different meaning for different types of assertions.

- A precondition requires that a condition to be verified in the beginning of the execution of the method. Hence, besides the conditional waiting involved in a concurrent precondition, it is necessary to ensure also that the objects involved in that condition can only be modified by the method itself (i.e. an eventual external synchronization also applies).

- For postconditions the same situation does not apply because the code it asserts is terminating.

- In this subject, invariants behave like a precondition in the beginning of (public) methods, and like postcondition at the end.

Flux Control and Concurrency

Flux Control and Concurrency

- In imperative programming the flux control of programs is determined by the following instructions:
  - sequence;
  - conditional instruction;
  - iterative instruction;
  - functions.

- A sequential programs the meaning of each one of these mechanisms is very clear. Lets take as an example the conditional instruction:

```
if(CONDITION) {

    COMMANDS;

}
else {

    COMMANDS;

}
```

```
if(CONDITION) {
    assert CONDITION;
    COMMANDS;
}
else {
    assert !CONDITION;
    COMMANDS;
}
```

Flux Control and Concurrency

- Thus, it makes no sense to select code with a conditional (or iterative) instruction if the language does not ensure the contracts (meaning) attached to those instructions.
- The problem arises when we consider the possibility of the selecting condition to be a concurrent boolean condition.
- That is, a condition whose value might depend on a thread other than the one execution the instruction.
- The semantics attached to each one of these elementary instructions is only kept if it is ensured that the condition value remains the same in the selected code block.

Flux Control and Concurrency

- So we reach the following rule:

> **Code selection instructions**
>
> The instructions that conditionally select code
> through concurrent conditions only make sense if
> the reservation (or an atomic alternative) of the
> shared objects present in the condition is atomically
> attached to the condition test.

- As a corollary we have that external synchronization is
  closely associated with the semantics of instructions that
  conditionally select code.

Flux Control and Concurrency: example

```
if (!stack.isEmpty()) {
    int elem = stack.top();
    stack.pop();
}
else
    System.out("Empty stack!!!");
```

```
synchronized(stack) {
    if (!stack.isEmpty())
    {
        int elem = stack.top();
        stack.pop();
    }
    else
        System.out("Empty stack!!!");
}
```

Internal synchronization schemes

Internal synchronization schemes: Monitor

- Force mutual exclusion in the use of all the object's public methods.

$$COA_P = \frac{1}{n}$$

$$i \in \{1, \ldots, n\}$$

$\cdots\cdots$: Monitor

$\longrightarrow$: Thread
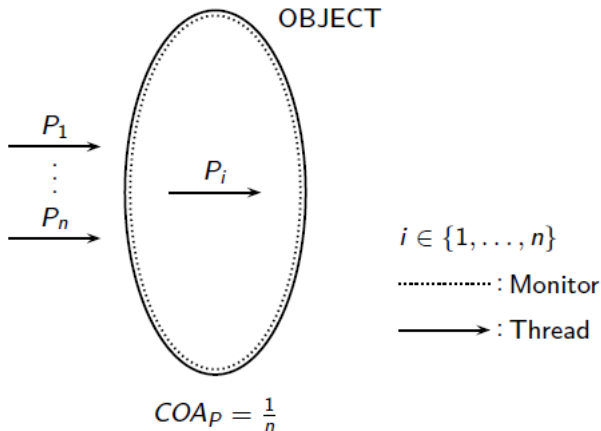
Internal synchronization schemes: Readers-writer exclusion

- Take advantage on the command-query separation;
- Commands are writers and queries are readers.
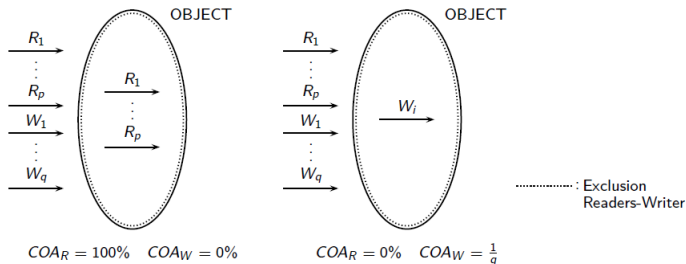
$$COA_R = 100\% \quad COA_W = 0\%$$

$$COA_R = 0\% \quad COA_W = \frac{1}{q}$$

⋯⋯⋯⋯ : Exclusion
Readers-Writer

Shared objects synchronization programming

Shared objects synchronization programming: requirements

- Correctness criteria:
  - Concurrent integrity of objects;
  - Total object covering.
- Support for the three aspects of synchronization:
  - internal (intra-object);
    - Necessary to ensure the internal correctness of the object (object usable only in its stable times).
    - Two synchronization schemes were already presented: monitor and readers-writer exclusion.
  - conditional;
    - Assumes a safe waiting for the desired condition.
    - It can be applied with condition variables (over locking schemes, such as monitors, readers-writer exclusion, or others).
  - external (inter-object).
    - "Exclusive" access to more than one operation on a shared object.
    - Implemented with a mutual exclusion scheme eventually with a mixed exclusion scheme (discussed in next lecture).

Shared objects synchronization programming

- In an object-oriented perspective, the implementation of shared objects only imposes the observation of the presented correctness criteria;

- As long as the criteria are met, any synchronization scheme can be used (from the simplest monitor alike scheme, until, eventually, lock-free schemes);

- Semantically all shared objects with a proper synchronization scheme are identical!

- Thus, it is easy to conclude that we have all the interest to separate, as clearly as possible, between the code that defines the logical sequential behavior of objects, from the synchronization code.

Shared objects synchronization programming

- The more common option is to put synchronization code together with logical business code inside the class (hard-coded).
- However, this option, has several problems:
    - It does not allow the reutilization of the class for purely sequential objects;
    - Mixes the synchronization logic with the business logic of the object.
    - Makes it very difficult to change the synchronization scheme.
- Thus, often it justifies to separate these two problems into different classes (programming of the object and synchronization programming);

Shared objects synchronization programming

- Synchronization programming can be done using the inheritance mechanism, redefining sequential methods and adding synchronization code;
- Alternatively, a client relation can be used to implement synchronization.
- These two alternatives are not equivalent from the point of view of OOP:
  - The first alternative although it directly uses fields and methods of the sequential parent class, makes it harder to provide a synchronization implementation for all objects that implement a given ADT (i.e. that are descendants of the sequential class);
  - The second, although it requires the declaration of a field that contains the reference to the sequential object, makes it possible to provide a synchronization implementation for a whole hierarchy of classes (that implement a given ADT). Hence, this is the preferred choice.

Shared objects synchronization: as a heir

```java
public class Stack<T> {
  public void push(T e) {
    Node<T> n =
      new Node<T>();
    n.e = e;
    n.next = top;
    top = n;
    size++;
  }

  public void pop() {
    assert !isEmpty();

    top = top.next;
    size--;
  }

  ...

  protected Node<T> top = null;
  protected int size = 0;

  protected class Node<T> {
    Node<T> next = null;
    T e;
  }
}
```

```java
public class SharedStack<T>
    extends Stack<T> {

  public synchronized void push(T e) {
    super.push(e);
    notifyAll();
  }

  public void pop() {
    assert !Thread.holdsLock(this) ||
           !isEmpty();
    synchronized(this) {
      try {
        while (isEmpty())
          wait();
        super.pop();
      }
      catch (InterruptedException e) {
        throw
  new UncheckedInterruptedException(e);
      }
    }
  }

  ...
}
```

Shared objects synchronization: as a client

```java
public abstract class Stack<T> {

  public abstract void push(T e);

  //@ requires !isEmpty();
  public abstract void pop();

  //@ requires !isEmpty();
  public abstract T top();

  public boolean isEmpty() {
    return size() == 0;
  }

  public abstract int size();
}
```

```java
public class SharedStack<T> {

  //@ requires stack != null;
  public SharedStack(Stack stack) {
    this.stack = stack;
  }

  public synchronized void push(T e) {
    stack.push(e);
    notifyAll();
  }

  public void pop() {
    assert !Thread.holdsLock(this) ||
           !stack.isEmpty();
    synchronized(this) {
      try {
        while (stack.isEmpty())
          wait();
        stack.pop();
      }
      catch (InterruptedException e) {
        throw
  new UncheckedInterruptedException(e);
      }
    }
  }
  ...
  protected final Stack<T> stack;
}
```

Abstract Synchronization

Subtype polymorphism inherent to OOP can also be applied to the programming of synchronization of shared classes:

```java
public abstract class SharedStack<T>
{

  //@ requires stack != null;
  public SharedStack(Stack stack) {
    this.stack = stack;
  }

  public abstract void push(T e);

  //@ requires !isEmpty();
  public abstract void pop();

  //@ requires !isEmpty();
  public abstract T top();

  public boolean isEmpty() {
    return size() == 0;
  }

  public abstract int size();

  protected final Stack<T> stack;
}
```

```java
public class MonitorStack<T>
  extends SharedStack<T> {

  public synchronized void push(T e) {
    stack.push(e);
    notifyAll();
  }

  public void pop() {
    assert !Thread.holdsLock(this) ||
           !stack.isEmpty();
    synchronized(this) {
      try {
        while (stack.isEmpty())
          wait();
        stack.pop();
      }
      catch (InterruptedException e) {
        throw
  new UncheckedInterruptedException(e);
      }
    }
  }

  ...
}
```