Lecture 05

Object-Oriented Concurrent Programming

Shared objects

Object-Oriented Concurrent Programming, 2019-2020

v3.3, 17-10-2019

Miguel Oliveira e Silva, DETI, Universidade de Aveiro

Contents

1 Shared objects synchronization programming

```
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();
}
```

⇒ Shared property made explicit.

⇒ Same public interface.

⇒ No inheritance relationship.

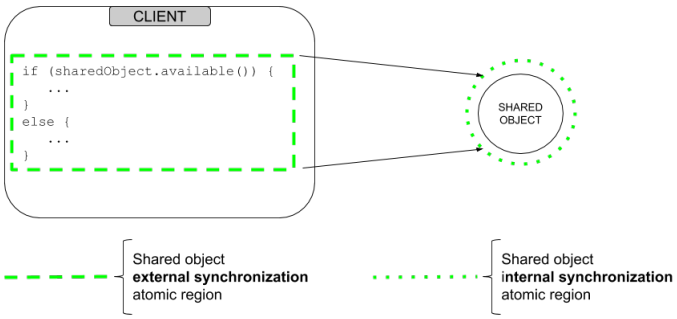
⇒ Concurrent assertions

```
public class SharedStack<T> {

  public SharedStack(Stack stack) {
    assert stack != null;
    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;
}
```

1. *Internal* synchronization
   ⇒ Object's ADT
2. *Conditional* synchronization
   ⇒ Concurrent assertions
3. *External* synchronization
   ⇒ Code selection with a concurrent condition (e.g. conditional instruction)

1.1 Programming External Synchronization

- A complete support for using shared objects requires programming all synchronization aspects: *internal*, *conditional*, and *external*.
- However, unlike internal synchronization, *external synchronization* does not have a well defined boundary.
- Its scope depends not of a simple modular entity such as an object (has happens with internal synchronization), but on client's code (for instance, as a result of a conditional instruction on a concurrent condition).



- Obviously, new client's may require unpredictable new blocks of external synchronization.
- Since its the shared object's responsibility to ensure all three aspects of synchronization, a modular solution must be devised for this problem.
- To that goal, shared objects can be extended with external synchronization services (e.g. `grab` and `release`) that ensure an (external) atomic behaviour.
- The question is: in practice, what synchronization schemes can be used to this goal?

- As mentioned, unlike internal synchronization – which is encapsulated to object's modular boundaries – external synchronization occurs in clients and has no defined frontiers – any sequence of object method invocation is possible.
- Because all synchronization aspects are required to be mutually consistent with atomic alike semantics (except when a fault occurs), we are almost forced to use the more restricted synchronization scheme: *mutual exclusion*;
- Although external synchronization can also be programmed with a software transactional approach, such an approach is restricted to repeatable operations (operations that might be transparently repeated, when a transaction fails).
- So, the simplest solution in practice, is to use mutual exclusion for external synchronization.

- Therefore, a new probleam arises: the solution must be compatible with the other two synchronization aspects.
- Of course, a possible solution is to restrict internal and external synchronization to the *same mutual exclusion* scheme.

Programming External Synchronization with `synchronized`

```
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();
}
```

```
SharedStack<Double> stack = ...;

synchronized(stack) {
   if (stack.size() >= 2) {
      double v1 = stack.top();
      stack.pop();
      double v2 = stack.top();
      stack.pop();
      stack.push(v1 + v2);
   }
}
```

```
public class SharedStack<T> {

  public SharedStack(Stack stack) {
    assert stack != null;
    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;
}
```

Programming External Synchronization with explicit mutex

```
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();
}
```

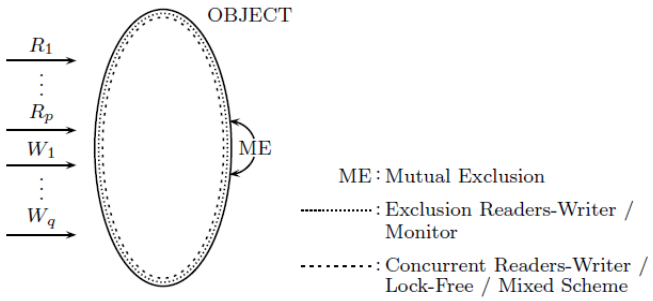
```
SharedStack<Double> stack = ...;
stack.grab(); try {
   if (stack.size() >= 2) {
      double v1 = stack.top();
      stack.pop();
      double v2 = stack.top();
      stack.pop();
      stack.push(v1 + v2);
   }
}
finally { stack.release();}
```

```
public class SharedStack<T> {
  public SharedStack(Stack stack) {
    assert stack != null;
    this.stack = stack;
    mtx = new Mutex(true);
    emptyCV = new MutexCV(mtx);
  }
  public void push(T e) {
    mtx.lock(); try {
      stack.push(e);
      emptyCV.broadcast();
    }
    finally { mtx.unlock(); }
  }
  public void pop() {
    assert !mtx.lockIsMine() ||
           !stack.isEmpty();
    mtx.lock(); try {
      if (!mtx.lockIsMine())
        while (stack.isEmpty())
          emptyCV.await();
      stack.pop();
    }
    finally { mtx.unlock(); }
  } ...
  public void grab() {mtx.lock();}
  public void release() {mtx.unlock();}
  protected final Mutex mtx;
  protected final MutexCV emptyCV;
  protected final Stack<T> stack;
}
```

1.1.1 Group mutual exclusion

- If external synchronization is (almost) restricted to mutual exclusion are we required to use the same scheme for internal synchronization?
- The answer is no, mixed synchronization schemes are possible. A separation of internal and external synchronization schemes is possible by using a mixed synchronization scheme named: *group mutual exclusion*.
- The basic idea is to use two groups, which mutually exclude themselves: i.e. group one excludes the use of group two and *vice-versa*;
- One group is applicable to object utilization that requires only internal synchronization, and the other for external synchronization uses.
- In what concerns the mutual exclusion scheme, method executions within each group work without exclusion; exclusion applies only when a group change is required.

- That is, a complete shared object synchronization uses first a group mutual exclusion scheme, and then inside each group, specific schemes are used for internal and external synchronization (the latter restricted to thread mutual exclusion).



- With the introduction of this new synchronization scheme, the programming complexity in the synchronization of shared objects increases raising new challenges.
- One of them is conditional synchronization!
- The usage of a group exclusion scheme pushes the object synchronization boundary to this scheme (specific internal and external synchronization schemes apply only after group exclusion).
- Thus, conditional synchronization is required to be bound to the group exclusion scheme (instead of the internal exclusion scheme).
- This difficulty is also an opportunity because it enables a common solution for conditional synchronization, both to internal and external synchronization, regardless of the choice for internal synchronization (it may even be a lock-free scheme).

- Nevertheless, it must be mentioned that since synchronization of shared objects is not automatic (as it would be in a high level concurrent language), in practice a programmer should always start by using a simple synchronization scheme (e.g. native synchronization or a simple mutex explicit lock, as exemplified).

1.2 Discussion

The challenge for abstract synchronization programming of shared objects is huge. Not only is necessary to support the three synchronization aspects – internal, external, and conditional – but also ensure that the three must peacefully coexist without having to be limited to the more restrictive synchronization scheme (mutual exclusion utilization). Being the simplest, this scheme is also the more restrictive for the concurrent use of objects (in fact, it sequentializes its use), and it may pose liveness problems in the program.

1.2.1 External synchronization

In the case of external synchronization few implementation alternatives exist. Up to the moment[1] the only safe and practical approach is a mutual exclusion scheme. It is practical because it does not force a detailed knowledge of the implementation of shared objects (which would break encapsulation). It is safe because it prevents race conditions in the access of shared objects (it can, of course raise liveness problems to the program).

The question that naturally is raised is if the choice of this scheme for external synchronization determines the choice for internal synchronization. It was showed that it has not to be like this, because we can safely use group exclusion scheme to alternate between this external synchronization scheme and any other that is being used for internal synchronization.

1.2.2 Conditional synchronization

However, this implementation for external synchronization raises a new problem. We know that there is the need for a conditional synchronization (waiting point) whenever there exists an assertion involving a concurrent condition (in the runtime context in which it is being tested). In particular, preconditions in shared object methods are many times conditional synchronization points. To be safe, the verification of this condition is required to be done *inside* the chosen internal synchronization scheme (whatever it is), but the waiting process must be *outside* the group exclusion scheme.

A natural implementation for this synchronization aspect uses condition variables, which are very efficient on the use of CPU, and, when properly used, are also a safe signal communication. For a peaceful coexistence of the three synchronization aspects, we need condition variables attached to the internal synchronization scheme that atomically also release the group exclusion synchronization.

1.2.3 Joint synchronization

Another practical problem posed by the implementation of abstract synchronization lies in the fact that the group exclusion synchronization have to be atomically attached not only to external synchronization (mutex), but also to internal synchronization. A failure to an external reservation of the object implies that waiting for this external resource cannot be inside the exclusion group, because it would disallow the access by threads belonging to other exclusion groups.

1.3 Implementation

In the supporting material, it is given a complete example of abstract synchronization applied to a queue data structure.

2 Shared objects: transitivity

Clearly, there is the need to identify shared objects in a program in order to avoid concurrent uses of sequential objects. These objects, referenced in a program by typed entities (class fields, method arguments and result, local variables) must be consistently synchronized and, preferably, referenced by typed entities that *explicitly* show that sharing property (shared classes).

It is intended in this way to avoid concurrent access to sequential objects. But will it avoid entirely?

The public interface of a shared class generally contains references to other types (in the method arguments and result, assuming the desired nonexistence of public fields). Hence, it is raised the problem of *transitivity* in the sharing property to the types that are part of the public interface of a shared class. Lets take a closer look at this problem by first stating the ultimate goal in this process:

Sharing of sequential objects forbidden: In the construction of concurrent programs it must be ensured that sequential objects are used by no more than one thread.

[1] There is a possible alternative using software transactions.

2.1 Primitive types

In `Java` there are two groups of types: primitive and reference types. Primitive types are characterized by not having a direct correspondence with a class and, mainly, by the *copy* semantics of assignment (either by directly by assignment instruction, or in argument passing). That is, in Java it is not possible to change a variable of a primitive type through another variable (in `C` and `C++` this is possible due to the existence of passing variables by reference). Hence there is no risk of sharing primitive typed entities[2]. Considering the impact to shared classes, this implies in the *non-transitivity* of the sharing property to the primitive types entities listed in the public interface.

Transitivity of primitive types: There is no sharing transitivity for primitive typed entities that are part of the shared object's interface (Abstract Data Type).

2.2 Reference types

Entities of reference types only points to objects (when not referencing `null`). That is, they are not not objects, but a (possible) reference to them. Hence, unlike primitive types, it is possible to observe and/or change the same objects through different reference typed entities in a program. This situation is termed *aliasing* and implies transitivity in the applicable properties of the objects.

Thus, whenever an object can be modified by more than one thread, it has to be a shared object (and properly and consistently synchronized).

Transitivity of reference types: There is transitivity for reference typed entities that are part of the shared object's interface (Abstract Data Type).

It should be note, that this need to propagate sharing properties applies to all reference typed entities related directly or indirectly to the class's public interface.

2.3 Immutable classes

A particular case of reference typed entities, is classes that construct (completely) immutable objects (e.g. `String`). In this case, being fulfilled the rules imposed by the language memory model, it can be considered that this objects and its types behave with the semantics of primitive types (copy without aliasing), hence no extra synchronization might be necessary.

However, it must be noted that the immutability of the object must be complete. That is, no mutable object can be referenced by the class's implementation.

Transitivity of immutable reference types: There is no sharing transitivity for reference immutable typed entities that are part of the shared object's interface (Abstract Data Type).

2.4 Shared objects due to transitivity

We have seen that a shared class has a transitive impact in all mutable reference types present in its interface. As a result, several consequences that must be taken into account:

- The need to create new shared classes;
- The need to impose a proper synchronization in the external use of those objects:
  - External and conditional synchronization if the object is referenced in the method's precondition;
  - Conditional synchronization if the object is referenced in another assertion;
  - Eventual external synchronization if the object is part of a condition that selects code (conditional or iterative instruction).

[2] We will see that the same thing happens with immutable primitive types