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

Shared Objects Synchronization

Shared Objects Synchronization


```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();
}
```

⇒ Shared property made explicit.

⇒ Same public interface.

⇒ No inheritance relationship.

⇒ Concurrent assertions

```java
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;
}
```

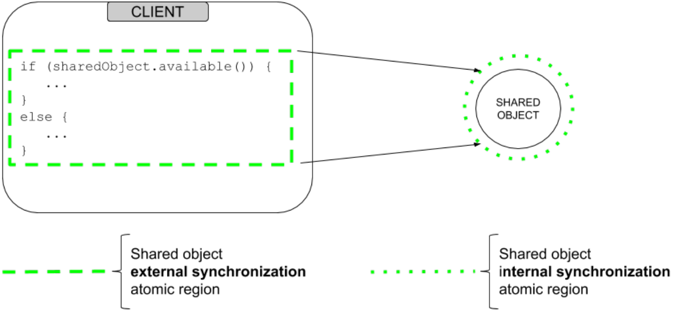
Shared Objects Synchronization (2)

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

Programming External Synchronization

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).



Programming External Synchronization

- 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?

Programming External Synchronization

- 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.

Programming 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;
}
```

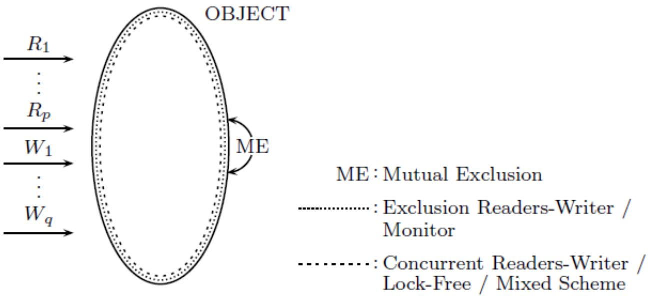
Group mutual exclusion

External Synchronization: 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.

External Synchronization: group mutual exclusion

- 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).



External Synchronization: group 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).

External Synchronization: group mutual exclusion

- 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).

Shared objects: transitivity

Shared objects: transitivity

- Clearly, there is the need to identify shared objects in a program in order to avoid concurrent uses of sequential objects.
- Is it enough?
- The public interface of a shared class usually contains references to other types (in the method arguments or in their result).
- Hence the problem of transitivity in the sharing property of objects is raised.

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.

Shared objects: transitivity of primitive types

- `Java` separates types in two groups: primitive and reference types.
- Primitive types have no direct correspondence with classes and have a *copy* assignment semantics.
- On the other hand, Java does not support passing variables by reference.
- Hence, it is not possible to change primitive type entities through another variable.

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).

Shared objects: transitivity of reference types

- Entities of reference types only points to objects (they are not objects).
- Hence, 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.

Transitivity of reference types

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

Shared objects: transitivity of reference types to immutable classes

- An important particular case is classes that create *immutable* objects (e.g. `String`).
- The assignment semantics immutable reference typed entities is similar to primitive types.
- In this case, if Java's memory model rules are fulfilled, we can consider that these objects and its types entities behave with primitive type semantics (no *aliasing*).

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).

- Watch out for the verification of a complete direct and indirect object immutability!

Shared objects due to transitivity

A shared class has a transitive impact in all the mutable reference types present in its interface. From here, it may result:

- 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).