Contracts and Threading Guidance

When writing contracts on methods invoked in multi-threading contexts, one has to be careful not to define contracts on shared unprotected state that are too strong, namely that don’t hold for certain thread interleavings. This note provides guidance on how to write contracts in these situations.

# Assumptions

For contracts involving threading issue, the Monitor class (or related class) in the BCL must provide a predicate that enables testing whether a certain lock is held. We assume the following method for this purpose:

**public** **static bool** IsHeld(**object** lockObject);

# Running Example

To illustrate the concepts, we use a fragment of some collection class that has the following methods

**public int** Count { **get**; }

**public object** Pop();

**public void** Push(**object** data);

# Classification

We classify methods into the following categories and analyze the impact of pre- and post-conditions in each category. Object invariants are discussed in a separate section.

1. ***Methods that do not perform internal synchronization on shared state***.  
   If such methods access shared state, the caller has to guarantee proper locking around them. For such methods, pre-conditions and post-conditions can be written that may involve such shared state without further complication, as these conditions will be evaluated within the same locking scope as the rest of the method.

For the example, this means we can have contracts of the following form, expressing both the pre-condition on Pop to not pop from empty collections, and the post condition on Push that there is now one more element in the collection:

**public** **object** Pop() {

Contract.Requires( Count > 0 );

…

}

**public void** Push(**object** data) {

Contract.Ensures( Count == Contract.Old(Count) + 1 );

…

}

1. ***Methods that perform internal synchronization on a caller visible lock.***Such methods may be called in two possible contexts:
   * From a context where the protecting lock is already held
   * From a context where the protecting lock is not held

The contracts written on the method must be useful in both these contexts. For pre-conditions, this implies that they *cannot* refer to state protected by the lock, as callers not holding the lock cannot guarantee such preconditions reliably (another thread could modify the shared state just prior to the method acquiring the protecting lock, but after the caller has established the pre-condition). In our example, this means that there is some lock object associated with the collection, and it is accessible to clients via a property, e.g.,

**object** SyncRoot { **get**; }

No, Pop *cannot* have the precondition Count > 0, as the following code could still fail due to another thread calling Pop in-between the test and the Pop:

**if** (c.Count > 0) { **object** o = c.Pop(); … }

This means that Pop cannot have the obvious pre-condition in such method implementations and must instead fail at runtime if the collection is empty. This points to a general problem with synchronized methods, namely, that the caller/callee contract really can only be followed if the caller performs the synchronization. This can be made explicit by changing the method implementation to be caller synchronized and adding the following pre-condition:

**public** **object** Pop() {

Contract.Requires( Monitor.IsHeld(this.SyncRoot) );

Contract.Requires( Count > 0 );

…

}

For post-conditions on such methods, we can do slightly better than on pre-conditions in the case that the caller actually holds the lock. Any post-condition on the shared state can be guaranteed to hold if the caller holds the lock protecting the shared state:

**public void** Push(**object** data) {

Contract.Ensures( !Contract.Old(Monitor.IsHeld(this.SyncRoot)) ||

Count == Contract.Old(Count) + 1 );

…

}

This post-condition is essentially an implication, saying that if the caller holds the lock in the pre-state, then the Count is one larger than the old Count on return.

1. ***Methods that perform internal synchronization on non-caller visible locks.***Such methods cannot be called in contexts where the caller already holds the protecting lock (as it is not caller visible). Thus, methods *cannot* have pre- or post-conditions involving shared state.

# Object Invariants

Object invariants are typically written over internal state that may be shared if the object is shared. For externally synchronized methods, object invariants pose no problem, as the invariant is evaluated in the same locking scope as the body of the method.

For self-synchronizing methods, these invariant checks should be performed while the lock is still held. Unfortunately, without declarations describing which lock protects the object state, the contract checking instrumentation cannot determine where to insert the object invariant checks to have them be performed reliably. There are three ways to deal with this situation:

1. Programmers avoid using object invariants on classes with self-synchronizing methods.
2. Programmers annotate self-synchronizing methods with an attribute [NoObjectInvariant] that instruct the rewriter to not instrument the object invariant check in such methods.
3. Object invariants on shared state are written as implication on the lock state, e.g.,

Contract.Invariant( !Monitor.LockHeld(OurLock) || \_count >= 0 );

This would allow calls in contexts where the caller holds the lock to check the invariant. For non-caller visible locks, it won’t do any good though. In that case, the implementer could call the ObjectInvariant checking routine directly from the implementation at points where the protecting lock is still held.