**Exclusion 🡪 Synchronization 🡪 Resource Ordering**

**2.2.6 Resource Ordering**

The need to preclude or recover from deadlocks and other liveness failures motivates the use of other

exclusion techniques presented in this chapter. However, one simple technique, *resource ordering* can

be applied to classes such as Cell without otherwise altering their structure.

The idea behind resource ordering is to associate a numerical (or any other strictly orderable data

type) tag with each object that can be held in a nested synchronized block or method. If

synchronization is always performed in least-first order with respect to object tags, then situations can

never arise in which one thread has the synchronization lock for x while waiting for y and another has

the lock for y while waiting for x. Instead, they will both obtain the locks in the same order, thus

avoiding this form of deadlock. More generally, resource ordering can be used whenever there is a

need to arbitrarily break symmetry or force precedence in a concurrent design.

In some contexts (see for example § 2.4.5), there may be reasons to impose some specific ordering

rules surrounding a set of locks. But in others, you can use any convenient tag for lock-ordering

purposes. For example, you may be able to use the value returned by

System.identityHashCode. This method always returns the default implementation of

Object.hashCode, even if a class overrides the hashCode method. While there is no guarantee

that identityHashCode is unique, in practice run-time systems rely on codes to be distinct with

a very high probability. To be even safer about it, you could override method hashCode or

introduce another tag method to ensure uniqueness in any classes employing resource ordering. For

example, you could assign each object a sequence number using one of the classes in § 2.2.4.

One further check, *alias detection,* can be applied in methods using nested synchronization to handle

cases in which two (or more) of the references are actually bound to the same object. For example, in

swapValue, you can check whether a Cell is being asked to swap with itself. This kind of check

is strictly optional here (but see § 2.5.1). Synchronization lock access is per-thread, not per-invocation.

Additional attempts to synchronize on already held objects will still work. However, routine aliaschecking

is a useful way to forestall downstream functionality, efficiency, and synchronization-based

complications. It may be applied before using synchronization surrounding two or more objects unless

they are of distinct, unrelated types. (Two references of two unrelated declared types cannot possibly

be referring to the same object anyway, so there is no reason to check.)

A better version of swapValue, applying both resource ordering and alias detection, can be written

as:

public void swapValue(Cell other) {

if (other == this) // alias check

return;

else if (System.identityHashCode(this) <

System.identityHashCode(other))

this.doSwapValue(other);

else

other.doSwapValue(this);

}

protected synchronized void doSwapValue(Cell other) {

// same as original public version:

long t = getValue();

long v = other.getValue();

setValue(v);

other.setValue(t);

}

As a minor efficiency tweak, we could further streamline the code inside doSwapValue first to

acquire the necessary locks, and then directly access the value fields. This avoids a self-call to a

synchronized method while already holding the required lock, at the minor expense of adding

lines of code that would need to be changed if the nature of the fields were ever modified:

// slightly faster version

protected synchronized void doSwapValue(Cell other) {

synchronized(other) {

long t = value;

value = other.value;

other.value = t;

}

}

Note that the lock for this is obtained via the synchronized method qualifier, but the lock for

other is explicitly acquired. A further, very tiny (perhaps nonexistent) performance improvement

might be obtained by folding the code in doSwapValue into swapValue, remembering to

acquire both locks explicitly.

Lock-ordering problems are by no means restricted to methods using nested synchronization. The

issue arises in any code sequence in which a synchronized method holding the lock on one object in

turn calls a synchronized method on another object. However, there is less opportunity to apply

resource ordering in cascaded calls: In the general case, one object cannot know for sure which other

objects will be involved in downstream calls and whether they require synchronization. This is one

reason that deadlock can be such a hard problem in open systems (see § 2.5) when you cannot release

synchronization during calls (see § 2.4.1).