Reentrant locking

Wim H. Hesselink (whh582)

March 4, 2022

1 Introduction

A reentrant lock, also called a recursive mutex, is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. It offers slightly more functionality than an ordinary lock, in that it allows the owner of the lock repeated calls of the lock. The ownership ends when the owner makes a matching number of calls to unlock.

As with an ordinary lock, when it is free, the first thread that calls the lock becomes the owner. When the lock is owned by a thread, other threads that call the lock will block and wait until the lock is free again.

The problem of locking is known as mutual exclusion. Following its introduction by E. W. Dijkstra [1], it can be described as follows. Given is a system of concurrent threads that each execute a loop of the form

```
loop of thread $p$ :
    $NCS$ ; $lock()$ ; $CS$ ; $unlock()$
end loop .
```

NCS stands for the *noncritical section*, a program fragment that need not terminate. *CS* stands for the *critical section*. The problem is to implement the procedures *lock* and *unlock* in such a way that there is never more than one thread in *CS*. Let us call this the *looping* specification (L).

Specification (L) is quite satisfactory for many purposes. It is, however, inadequate for specifying reentrant locking as a generalization of locking, because the setting precludes multiple calls of *lock*. The definition of mutual exclusion must therefore be reconsidered before reentrant locking can be treated.

Comparable specifications of locks and reentrant locks are proposed in Section 2. Section 3 briefly sketches the role of reentrant locks for synchroonization in the programming language Java. Section 4 discusses how an ordinary lock can be converted into a reentrant one.

Acknowledgement. I am grateful to Dave Dice, who proposed the problem and later pointed to the difficulties for the implementation.

2 Specifying locks and reentrant locks

For ordinary locking, the following specification (3R) is proposed. The lock has a state function `mu` of type $thread \cup \{\bot\}$, where $\bot$ stands for not-a-thread. Initially $\texttt{mu} = \bot$. Informally speaking, the value of `mu` is the thread that owns the lock. For the sake of the specification, the functions *lock* and *unlock* get the identifier of the calling thread as an argument. There are three requirements:

1. The procedures *lock* and *unlock* satisfy the wait-free Hoare triples:

$$\{\texttt{mu} = \bot\} \quad \mathit{lock}(p) \quad \{\texttt{mu} = p\} \ ,$$
$$\{\texttt{mu} = p\} \quad \mathit{unlock}(p) \quad \{\texttt{mu} = \bot\} \ .$$

2. Any call *lock*(p) with the precondition $p \neq \texttt{mu} \neq \bot$ blocks, and thread p waits until $\texttt{mu} = \bot$ holds.

3. If *lock*(p) is called with precondition $\texttt{mu} = p$, or *unlock*(p) is called with precondition $\texttt{mu} \neq p$, the result is undefined (an exception can be raised).

The specifications (L) and (3R) are equivalent. Indeed, if the lock satisfies (L), one can introduce a ghost variable `mu`, and can insert an assignment $\texttt{mu} := p$ as a final command of *lock*, and an assignment $\texttt{mu} := \bot$ as an initial command of *unlock*. This implies the requirements 1 and 2. Requirement 3 makes explicit that the setting of (L) does not allow calls as mentioned in this requirement. Conversely, if a lock satisfying (3R) is used in the loop of (L), requirements 1 and 2 imply mutual exclusion.

Specification (3R) specifies the unfair lock. One can introduce fairness for the lock by means a set or queue of waiting threads with an indication of the waiting thread(s) preferred for the next ownership. This matter of fairness is orthogonal to the aspect of reentrance. It can therefore be ignored for the present purposes.

The reentrant lock is specified by means of two state functions `mu` and `level`. Function `mu` has the same type, role, and initialization as above. Function `level` holds a natural number, initially 0. If positive, it holds the number of times thread `mu` must call *unlock* to open the lock. Let us use the names *re-lock* and *re-unlock* for the two procedures of a reentrant lock. There are four requirements:

1. $\texttt{mu} \neq \bot \quad \equiv \quad \texttt{level} > 0$.

2. The procedures *re-lock* and *re-unlock* satisfy the wait-free Hoare triples:

$$\{\texttt{mu} = \bot\} \quad \textit{re-lock}(p) \quad \{\texttt{mu} = p \ \wedge \ \texttt{level} = 1\} \ ,$$
$$\{\texttt{mu} = p \ \wedge \ \texttt{level} = k\} \quad \textit{re-lock}(p) \quad \{\texttt{mu} = p \ \wedge \ \texttt{level} = k+1\} \ ,$$
$$\{\texttt{mu} = p \ \wedge \ \texttt{level} = 1\} \quad \textit{re-unlock}(p) \quad \{\texttt{mu} = \bot\} \ ,$$
$$\{\texttt{mu} = p \ \wedge \ \texttt{level} = k > 1\} \quad \textit{re-unlock}(p) \quad \{\texttt{mu} = p \ \wedge \ \texttt{level} = k-1\} \ .$$

3. Any call *re-lock*(p) with the precondition $p \neq \texttt{mu} \neq \bot$ blocks, and thread p waits until $\texttt{mu} = \bot$ holds.

4. If *re-unlock*(p) is called with precondition $\texttt{mu} \neq p$, the result is undefined (an exception can be raised).

It follows that the reentrant lock behaves as an ordinary lock as long as no owner of the lock calls *re-lock*.

3 Synchronization in Java

The programming language Java has a keyword **synchronized**, see Java Language Specification, Chapter 17. It has synchronized statements and synchronized methods. In either case, there is an associated lock and the body of the statement or method is executed only after the acting thread has obtained the lock. When the body has terminated, the associated *unlock* is called. As a synchronized method may call another method synchronized by the same lock, this description requires the lock to be reentrant.

If the lock is reentrant, correctness is implied by the observation that the following four assertions are equivalent:

$$\begin{aligned} & \texttt{mu} = \bot \\ \equiv\quad & \texttt{level} = 0 \\ \equiv\quad & \text{the number of performed } \textit{lock} \text{ operations equals the number} \\ & \quad \text{of performed } \textit{unlock} \text{ operations of the lock} \\ \equiv\quad & \text{every synchronized action of the lock has terminated.} \end{aligned}$$

4 Implementation

There are several ways to convert an ordinary lock into a reentrant one. The simplest way is to use shared variables `mu` and `level`, and to closely follow the specification. Jonas Oberhauser proposed the following variation, which may be somewhat more efficient. It uses shared variables `mu` and `lev`:

```
initially: $\texttt{mu} = \bot \ \wedge\ \texttt{lev} = 0$ .

re-lock(p) =
    if $p = \texttt{mu}$ then lev++
    else
        lock(p) ;
        mu := p
    endif .

re-unlock(p) =
    assert($p = \texttt{mu}$) ;
    if $\texttt{lev} = 0$ then
        mu := $\bot$ ;
        unlock(p)
    else
        lev--
    endif .
```

The state function `level` is given by the equality

$$\texttt{level} = \texttt{lev} + (\texttt{mu} = \bot\ ?\ 0 : 1)\ .$$

It is then straightforward to show that, if the pair *lock*, *unlock* satisfies the specification of an ordinary lock, the pair *re-lock*, *re-unlock* satisfies the specification of a reentrant lock.

There is, however, one subtle complication. The variable `mu` is written always under protection of the lock, but it can be read by a thread p when p does not own the lock. In this case, the test $p = \texttt{mu}$ should return false. If it does not, somehow the reading of `mu` returns the value p, either out of the past of the lock, or out of the blue. It seems that, with this mixed access mode of `mu`, such a false positive cannot be ruled out completely, because the semantics of locking do not cover these cases. So, the result may depend on processor, platform, threading system, and programming language.

If such false positives cannot be precluded, there is the alternative to give every thread a private set (list) *owning* of lock identifiers and replace the test $p = \texttt{mu}$ by the test $\textit{lock} \in \textit{owning}_p$. This approach would be preferred for Java, because its virtual machine does track the locks held by a thread in order to release them when the thread dies or throws an exception out of a synchronized region. In practice most threads hold only a few locks, so the lists are short and not bad to search.

References

[1] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8:569, 1965.