A Crossing with Java Threads and POSIX Threads

Wim H. Hesselink, 15th October 2001

Abstract

The primitives for Java threads and POSIX threads are compared by means of a simulation of cars at a crossing. These cars have to be synchronized in such a way that at every moment only cars in one direction proceed. The initial solution is based on compound await statements. This solution is subsequently implemented with Java threads and POSIX threads. These two thread formalisms differ: POSIX threads can wait at condition variables of a greater generality than available in Java, but the corresponding queues may be leaky.

Keywords concurrency, thread, Java, POSIX threads, await statement, program correctness

1 Introduction

In our favorite design method for concurrent algorithms, at some point the threads or processes are synchronized by means of atomic synchronization statements of the form:

(0) $\quad \langle\, \textbf{await } B \textbf{ then } S \,\rangle$.

This is the command to wait until predicate B holds and then to execute command S without interference, cf. [1, 2]. If B is identically true, the **await** statement reduces to the atomic statement $\langle\, S \,\rangle$.

Programming languages never offer construct (0), since a general implementation is regarded as too costly. Some programming languages, e.g., Modula-3 and Java, support threads with some synchronization primitives. The POSIX committee for the standardization of UNIX has also specified a thread package with synchronization primitives. In this note, we compare the synchronization primitives offered by Java with those of the POSIX thread standard.

We focus on an example that is easy enough to deal with and hard enough to show the differences. The example is a simulation of traffic at a crossing. At the crossing, traffic is coming from different directions. It is modelled by a number of threads that represent cars and that repeatedly execute a procedure *cross* with an argument *dir* for the direction. To prevent accidents, it must be disallowed that cars from different directions are crossing concurrently.

This problem is treated in [5] as the Congenial Talking Philosophers problem. The case of two directions is the narrow bridge problem of [3], p. 294, a Java solution of which is given in [6] 7.4. It is also generalization of the classical problem of readers and writers, when we associate all readers with a single direction, say 0, and every writer with a unique private nonzero direction.

The paper [5] gives a solution based on busy waiting and atomic read-write registers. We aim at simpler solutions based on stronger primitives. In Section

2, we give an abstract solution based on the await construct (0). We then turn to implementations of this solution, depending on the available synchronization primitives. In Section 3, we give an implementation in Java. Section 4 contains an implementation with POSIX threads. In Section 5, we draw some conclusions.

2 The abstract problem and a solution

We formalize the problem by stating that all threads execute

(1)
while *true* **do**
0: *entry*(*dir*) ;
1: *cross*(*dir*) ;
2: *depart*() ;
3: *other activity* ;
od .

The problem is to implement *entry* and *depart*. Procedure *entry* may involve waiting. Procedure *depart* may need exclusive access to shared data but must not involve conditional waiting. It is given that *cross* terminates; the *other activity* need not terminate.

We use $q.dir$ for the direction dir of thread q. The safety requirement is that cars can only cross concurrently when using the same direction, i.e. for all q and r,

(Safety) q **at** 1 $\wedge$ r **at** 1 $\Rightarrow$ $q.dir = r.dir$.

The progress requirement is that, whenever there are n cars registered to cross along direction dir, eventually at least n cars will cross along dir. Finally, it must be allowed that cars with the same direction cross concurrently.

In this section we give an abstract solution based on await construct (0). In order to allow concurrent access for cars from the same direction, we introduce a shared variable `cur` for the direction currently allowed to cross. All traffic coming from directions other than `cur` is suspended. When the suspended cars from the current direction have passed the crossing and there are suspended cars for other directions, a new current direction is chosen.

We introduce a boolean shared variable `free` to indicate the absence of competing cars, and integer shared variables `ncc` for the number of crossing cars and `due` for the number of cars from the current direction that will be allowed to cross. The latter serves to enable a fair treatment of the various directions. Initially, `free` is true and `ncc` and `due` are 0.

Now *entry* is refined by

entry1(*dir*) =
⟨ **await** `free` $\vee$ (*dir* = `cur` $\wedge$ `due` > 0) **then**
`cur` := *dir* ; `free` := *false* ;
`due --` ; `ncc ++` ⟩

and *depart* is refined by

depart1() =
⟨ `ncc --` ;
if `ncc` $= 0 \;\wedge\;$ `due` ≤ 0 **then**
if possible choose `due` > 0 and new direction `cur` such that
`due` cars with direction `cur` are registered at *entry*
or else `free` $:=$ *true* **fi** ;
fi ⟩ .

The new direction must be chosen fairly, e.g., by choosing the new direction `cur` in a circular order (round robin). Note that cars with the same direction are allowed to cross concurrently.

Predicate (Safety) is proved as follows. It clearly follows from

(K0) $\quad q$ **at** $\{1,2\} \;\Rightarrow\; q.dir =$ `cur` .

We now prove that (K0) is an invariant. Execution of *entry1* establishes (K0) for the acting thread. Predicate (K0) is threatened when some thread $p \neq q$ chooses a new direction `cur` in *depart1* while q is at 1 or 2. This case does not occur, however, because of the invariant that `ncc` always equals the number of crossing cars:

(K1) $\quad$ `ncc` $= (\#\, x :: x$ **at** $\{1,2\})$.

Finally, predicate (K0) is threatened when $p \neq q$ executes *entry1* and q is at 1 or 2 and $dir.p \neq dir.q =$ `cur`. We then use (K1) and the invariant

(K2) $\quad$ `free` $\;\Rightarrow\;$ `ncc` $= 0$,

to conclude that `free` is false so that p cannot execute *entry1*. The verification of the invariants (K1) and (K2) is easy. This concludes the proof of (Safety).

For absence of deadlock of the crossing, it suffices to require that the crossing is always free when there are no cars crossing or about to cross:

(K3) $\quad$ `ncc` $= 0 \;\Rightarrow$ `free` $\vee$ `due` > 0 ,
(K4) $\quad$ `due` $\leq (\#\, x :: x.dir =$ `cur` $\;\wedge\; x$ **at** $0)$.

It is easy to see that (K3) and (K4) are invariants. Progress follows from the way `cur` is modified.

3 Synchronization with Java

We now let the cars be Java threads and we implement the atomicity and await statements in Java. Java offers so-called *synchronized* methods. When a thread has entered a synchronized method of an object, no other thread can execute any synchronized method of that object. Therefore, in order to implement the abstract solution of section 2, we introduce an object `synch` to synchronize the access to the shared variables `free`, `cur`, `ncc`, and `due`. So we implement these variables as instance variables of a shared object `synch`.

Java also offers primitives for thread suspension. When a thread calls `wait()` inside a synchronized method of an object, the thread is deactivated and put into the waiting queue of the object. Threads queued at an object can be awakened by calls of `notify` and `notifyAll` for this object. The call `notify()` awakens precisely one waiting thread (if one exists), whereas `notifyAll()` awakens all threads waiting at the object.

When all waiting cars would be waiting at the same object, at every notification all waiting cars would have to reexamine their guards. To avoid contention, we decide that cars for different directions shall wait at different objects (traffic lights). Before treating the traffic lights, however, we look at the question whether an entering car must wait.

Each entering car has to decide whether to stop at a traffic light or proceed to cross. This is decided in the method `mustStop` of `synch`. The object `synch` also registers how many cars are queued at every traffic light by means of a variable `count[]`, where `count[j]` holds the number of cars waiting at traffic light `j`. The decision and the actions of *entry* are thus implemented by

```
synchronized boolean mustStop (int dir) {
   if (free || (dir == cur && due > 0)) {
      cur = dir ; free = false ; due -- ;
      ncc ++ ; return false ;  // can proceed
   } else {
      count[dir] ++ ; return true ;  // must stop
   }
}
```

We let the directions be numbers j with $0 \leq j <$ `dirLim`. Command *depart* is now implemented in the method `depart2` of `synch`, given by

```
synchronized void depart2 () {
   ncc -- ;
   if (ncc == 0 && due <= 0) {
      int j = (cur + 1) % dirLim ;
      while (j != cur && count[j] == 0)
         j = (j+1) % dirLim ;
      if (count[j] > 0) letGo (j) ;
      else free = true ;
   }
}
```

The **while** loop is a circular search for the first direction with positive `count` value. This precludes starvation of directions.

It remains to implement `letGo`, but that depends on the implementation of the traffic lights. We use an array of traffic lights, i.e., one object for each direction. When a new direction is chosen, all cars waiting for that direction can be awakened.

We now have the problem that the cars that have passed `mustStop` with result `true` need not immediately be waiting at the traffic light. Therefore,

`letGo` must not only notify cars waiting at the traffic light, but it must even enable cars at the traffic light that have not yet entered the waiting queue.

We therefore apply a semaphore for every direction. Our semaphore is an object with an integer variable `val`, initially 0, with two methods

```
synchronized void up (int n) {
   val = val + n ;
   notifyAll () ;
}

synchronized void down () {
   while (val <= 0) {
      try { wait () ; }
      catch (InterruptedException e) {}
   }
   val -- ;
}
```

A call of `up(n)` with `n` ≥ 0 allows `n` threads to pass `down()`.

We introduce an array `light[dirLim]` of these semaphores and we implement method `letGo` of `synch` by

```
void letGo (int dir) {
   cur = dir ;
   due = count[dir] ;
   count[dir] = 0 ;
   light[dir].up(due) ;
}
```

Method `letGo` is not synchronized, since it is called (only) from the synchronized method `depart2`.

Command *entry* is implemented by

```
void entry2 (int dir) {
   while (synch.mustStop (dir)) light[dir].down() ;
}
```

The repetition in `entry2` is needed since the actions of *entry1* are done in `mustStop` when it returns false. In this way we get all invariants of Section 2. Repeated testing is also needed because of other cars competing for the current direction. Method `entry2` is not synchronized since it connects two synchronized objects `synch` and `light[dir]`.

It remains to verify that `depart2` always determines new values for `due` and `cur` according to the specification in *depart1*. Let `NQ(j)` be the number of threads at `down` of `light[j]`. Whenever a thread arrives there, it has incremented `count[j]` in `mustStop`. Whenever a thread passes `light[j]`, it decrements `light[j].val` in `down`. Since the sum of these numbers is kept invariant in `letGo`, we have the invariant

(Ja) $\quad$ `NQ(j)` $=$ `synch.count[j]` $+$ `light[j].val` .

This implies that, as required, method `letGo` always establishes `due` $\leq$ `NQ(cur)`. Moreover, if method `depart2` makes `free` true, all elements of `count` are zero, so that (Ja) implies that there are no threads blocked at the semaphores.

Remark. By using semaphores for the traffic lights, we avoid that all waiting threads have to test their guards when the direction changes. This improves the performance when there are many directions. The narrow bridge in [6] 7.4 can be simpler since it has only two directions.

4 An implementation with POSIX threads

The POSIX standard specifies *mutexes* with the main primitives `pthread_mutex_lock` and `pthread_mutex_unlock`, which are abbreviated here by `lock` and `unlock`. Enclosing a program fragment by `lock`(`m`) and `unlock`(`m`) for a mutex `m` has the same effect as encapsulating the fragment in a synchronized Java method. The `unlock` must be called by the thread that `lock`ed the mutex.

The POSIX standard provides *condition variables* for conditional waiting. We regard a condition variable `v` as a set of threads, which is initially empty. We only use the POSIX primitives `pthread_cond_wait` and `pthread_cond_broadcast`, abbreviated by `wait` and `broadcast`. These primitives are expressed as follows.

(2) $\quad$ `wait`(`v`, `m`) :
$\quad\quad$ $\langle$ `unlock` (`m`) ; `v` $:=$ `v` $\cup \{self\}$ $\rangle$;
$\quad\quad$ $\langle$ **await** *self* $\notin$ `v` **then** `lock`(`m`) $\rangle$.

Note that command `wait` consists of *two* atomic commands: one to start waiting and one to lock when released. Also, note that a thread must hold the mutex to execute `wait`. Command `broadcast`(`v`) releases all threads waiting in `v`.

(3) $\quad$ `broadcast`(`v`) : $\quad \langle$ `v` $:= \emptyset$ $\rangle$.

There is one additional complication: a thread waiting at a condition variable may wake up spontaneously. According to Butenhof, who was involved in the pthreads standard from its beginning, this is motivated as follows, cf. [4], p. 80: "On some multiprocessor systems, making condition wakeup completely predictable, might substantially slow all condition variable operations. The race conditions that cause spurious wakeups should be considered rare."

The possibility of spurious wakeups implies that, for every conditional call of `wait`, the condition must be retested after awakening.

The synchronization of a Java method at a certain object can be emulated by introducing a mutex `m` for the object and enclosing the method by calls of `lock(m)` and `unlock(m)`. The Java commands `wait` and `notifyAll` are emulated by the POSIX commands `wait` and `broadcast`, after the introduction of a condition variable `v` as an additional argument. Therefore, the Java crossing of Section 3 can be directly transferred to POSIX threads. The spurious wakeups

are harmless here, since in method `down` the waiting condition is tested again after awakening.

The separation of the roles of mutexes and condition variables enables a leaner solution. We can replace the semaphores of Section 3 by condition variables. We need only one mutex, `mut`, and an array of condition variables `cv[limDir]`. The entry protocol becomes:

```
void entry3 (int dir) {
   lock (mut) ;
   while (!free && (dir != cur || due <= 0)) {
      count [dir] ++ ;
      wait (cv[dir], mut) ;
      count [dir] -- ;
   }
   cur = dir ; free = false ;
   due -- ; ncc ++ ;
   unlock (mut) ;
}
```

Here we decrement `count` after waiting since the queues of the condition variables can be leaky.

The depart protocol is also guarded by mutex `mut` and otherwise similar to `depart2` in Section 3. Since the decrementation of `count` is moved to `entry3`, procedure `letGo` becomes

```
void letGo (int dir) {
   cur = dir ; due = count[dir] ;
   broadcast(cv[dir]) ;
}
```

Correctness relies on two invariants concerning `count[j]`. On the one hand, `count[j]` is an upper bound of the number of threads in the queue `cv[j]`:

(Po) $\quad \#\,\texttt{cv[j]} \leq \texttt{count[j]}$.

We cannot guarantee equality since there can be threads released from the queue that have not yet reacquired the mutex and decremented `count`. The invariant (Po) implies that all waiting queues are empty when `free` holds. This precludes forced starvation of cars.

On the other hand, we have the invariant that `count[j]` is a lower bound of the number of threads that need to retest the guard in the repetition of `entry3`. This implies that, always, at least `due` cars with direction `cur` are waiting at *entry* as required in *depart1*. As is shown in Section 2, this precludes deadlock of the crossing as a whole.

5 Concluding remarks

The POSIX primitives for synchronization are more flexible than those of Java. Indeed, the traffic lights were condition variables in pthreads and semaphores

in Java. In our view, the Java primitives encourage a better structuring and separation of concerns, but this is not really supported by the present case. The spurious wakeups of pthreads form a complication, that forced us to be more careful when counting the number of waiting threads.

References

[1] Andrews, G.R.: Concurrent Programming, principles and practice. Addison-Wesley 1991.

[2] Apt, K.R., Olderog, E.-R.: Verification of Sequential and Concurrent Programs. Springer V. 1991.

[3] Brinch Hansen, P.: Operating System Principles. Prentice Hall, Englewood Cliffs, 1973.

[4] Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley 1997.

[5] Joung, Y.-J.: Asynchronous group mutual exclusion. Distribut. Comput. **13** (2000) 189–206.

[6] Magee, J., Kramer, J.: Concurrency, state models & Java programs. Wiley (Chichester, etc.) 1999.