Unit 7: Blocking Synchronization
Vijay Saraswat

The purpose of this unit (Lectures 21, 23 – 25) is to introduce data-dependent blocking operations in X10.

1 Motivation

Consider situations in which multiple activities are performing mulitple operations on a shared resource. Unfortunately, some operations are *conditional*: they can succeed only if the resource is in a particular state.

Example 1.1 (Bounded Buffer) *Get and put operations. Get operation blocks if there is no more data to be read. Put operation blocks if the buffer is full.*

A good example of the use of bounded buffers is in streaming computations.

Example 1.2 (Asynchronous argument evaluation) *(futures)*

Aside Interestingly, it is rather hard to find examples of blocking synchronization in the real world. For instance, human beings dont remain blocked waiting for a particular action to happen for an indeterminate amount of time. (This would be detrimental to survival, e.g. the organism would not be able to respond to a higher priority, newly developing threat in the environment, a new predator arrives.)

Rather the method used is to try, wait for failure, try again and giving up on repeated failure (e.g. to abandon the goal, or try another way of achieving the goal). This can be modeled perfectly well with just `async`, `atomic` and `finish`.

2 Conditional atomic blocks

To express such patterns directly in X10 we find it convenient to introduce a *conditional atomic block*, `when (c) S` (where `c` is a `boolean` expression and `S` is a statement). The execution of this statement blocks until such time as `c` is true. Then, it executes `S` "instantly" (i.e. atomically with respect to all other activities).

```
1 while(true) {
2   atomic
3     if (c) {
4       S;
5       break;
6     }
7 }
```

2.1 Deadlock

Blocking leads to deadlock when multiple activities in a set are each waiting for another in the set to progress (sometimes called a "deadly embrace").

```
class Deadlock {
  var a:Int=0;
  var b:Int=0;
  def m() {
    async {
      when (a > 0)
        b=1;
    }
    when (b > 0)
      a=1;
  }
}
```

3 Programming idioms

Example 3.1 (Semaphore) *Here is a simple implementation of a semaphore:*

```
public class Semaphore {
  private var count:Int=0;
  public def this(v:Int) {
    count=v;
  }
  public def p() { // wait, acquire
    when (count > 0) count--;
  }
  public def v() { // signal, release
    atomic count++;
  }
}
```

A *latch* of type `T` is a simple data-structure that lives in one of two states: filled or empty. It is created in the empty state. It may be filled with a value of type `T`, but only once. An attempt to fill it once it has been filled raises an exception. Once it has been filled the value it is filled with can be retrieved.

Example 3.2 (Latch) *Here is the program.*

```
public class Latch[T] {
  private var data:T;
  private var filled:boolean=false;
  def this() {}
  def set(t:T) throws FilledException {
    atomic {
      if (filled) throw new FilledException();
      filled=true;
    }
    data=t;
  }
  def get():T {
    when (filled) return data;
  }
}
```

A *bounded buffer* of type `T` and size `n` is a data-structure that permits a producer to interact with a consumer without letting either get too far ahead of the other.

Example 3.3 *Here is the program.*

```
1  class Buffer[T] {
2    val data:Rail[T];
3    var slots:Int;
4    var r:Int=0;
5    var w:Int=0;
6    def this(n:Int, init:T){
7      slots=n;
8      data = Rail.makeVar[T](n, (int)=>init);
9    }
10   def put(t:T) {
11     when (slots > 0) {
12       slots--;
13       data(w)=t;
14       w++;
15       if (w==data.length) w=0;
16     }
17   }
18   def get():T {
19     var result:T;
20     when (slots < data.length) {
21       slots++;
22       result=data(r);
23       r++;
24       if (r==data.length)
25       r=0;
26     }
27     return result;
28   }
29 }
```

4 Language restrictions

The `when` statement in X10 is accompanied with many restrictions, imposed for efficiency of implementation.

- The body of `S` must be single place – no `at` allowed.
- The body of `S` must be non-blocking – no recursive `when` allowed.
- The body of `S` must be sequential – no `async` allowed.

Additionally, it is highly desirable that the body contain bounded number of operations.

5 Implementation notes

There are two principle implementation techniques for `when`: pessimistic and optimistic.

5.1 Pessimistic techniques

The pessimistic technique is to acquire a lock or a set of locks. The trick is to ensure that all access to mutable data within the `when` are governed by the same lock (or locks). If multiple locks are used they should be acquired in the same order. On acquiring the lock, a check is made to determine if the condition is satisfied. If it is, the body of the statement is executed (this can be done without any further blocking), the locks are released and statement execution terminates. If not, the activity is placed on a queue associated with the set of mutable variable accesses that were made in evaluating the condition.

Any writes to any of these variables are modified so that they signal the associated queue. Now whenever the asyncs in the queue are signalled, they again attempt to acquire the lock, determine if the condition is trur or not, as above.

Thus one can see the queue as an attempt to reduce the number of checks of the condition. In some special cases it should be possible for the compiler to generate efficient code that causes the asyncs in a queue to be woken up only when the condition is already satisfied.

5.2 Optimistic techniques

In these approaches the activity attempting to execute a `when` statement proceeds to do so, keeping track in a log of the mutable variables that it reads and writes. Imagine that there is a timestamp with each mutable variable; the log records the timestamp of the variable that was read or written. The writes are considered to happen in a private space and are not visible until commitment. Once the statement has completed, a decision is made whether to *commit* or *rollback*. The statement execution can commit only if no other activity has written to a variable that it has read (this can be determined by examining the version numbers). Otherwise the statement is rolled back and must be retried.

Once again, as above, attemps to retry can be reduced by keeping track of which variables were read when evaluating the condition and retrying only once one of the variables has changed.

6 Notes