Unit 6: Indeterminate Computation

Martha A. Kim

October 6, 2013

Introduction

Until now, we have considered parallelizations of sequential programs. The parallelizations were deemed "safe" if the parallel program were shown to be equivalent to the underlying sequential program via analysis of the parallel version's effect sets. We now turn our attention to a class of parallel programs that is considered "unsafe" by our previous definition. We call this class of parallel programs "indeterminate".

A program is deterministic on a given input if each memory location is updated with the same sequence of values in every execution. Determinism leads the program to always behave in the same way on the same input. Reasoning about and debugging programs that do not have this property – indeterminate programs – is significantly more challenging. In this unit we will reason about the correctness and progress conditions of indeterminate parallel programs.

Reasoning about interleavings

An interleaving is the order in which multiple simultaneous accesses to a single memory location are interleaved in time and applied to the memory location.

Consider the following two transactions applied to the same bank account. The body of the deposit thread (blue) and the withdrawal thread (red) each boil down to three instructions containing a load of and then a store to the single location in memory that holds the current account balance.

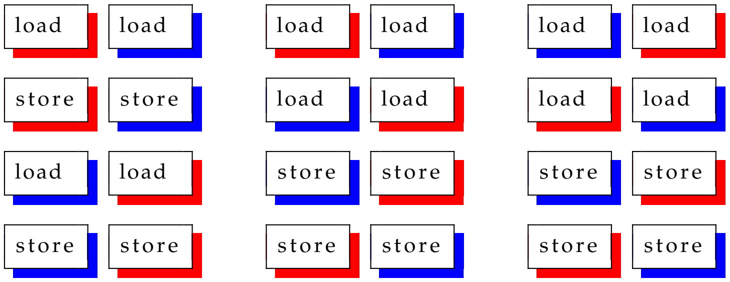
```
deposit (x) {
  balance += x;
}
```

```
withdraw (x) {
  balance -= x;
}
```

```
load balance $r0
addi $r0, $r0, x
store $r0, balance
```

```
load balance $r1
subi $r1, $r1, x
store $r1, balance
```

The correctness of the computation depends on the order in which these operations are applied to the shared memory location, in other words, the interleaving. Here we illustrate the six possible interleavings of the four memory operations in these two threads. Each thread dictates the order of its own instructions, but apart from that constraint the instructions may read and update memory in any order, resulting in the six potential interleavings shown below. The first two interleavings will compute a correct result, while the last four will not.



When these two threads are executed, as written, the user has no control over which of these six interleavings will occur. Sometimes the interleaving may be legal, other times starting from the very same inputs the result may not be. In order to restrict the possible interleavings to only the legal ones, it is up to the programmer to insert appropriate synchronization into their code.

Atomic operations

The problem with the above example is that, to operate correctly, the load and store in each thread must be executed atomically. Atomic operations are indivisible. They cannot be split by another thread.

This atomicity can be enforced at any level, either some runtime software system, or in the hardware itself.

Compare and swap

Most modern processors support one or two basic synchronization primitives. The first of these is the compare and swap operation. The compare and swap (CAS) operation takes three arguments: the memory location in question, an expected value, and an update value. If the location contains the expected value, it is overwritten with the update value. If the location does not contain the expected value, it will not be overwritten, and the CAS operation returns false.

In X10, `x10.util.concurrent` provides several atomic classes such as `AtomicInteger` and `AtomicReference` that support compare and swap. Similarly, in Java `java.util.concurrent.atomic` provides several basic classes (e.g., boolean, integer and reference classes) which have a compareAndSet() method [1] while C# supports the

[1] Sun Microsystems. Package java.util.concurrent.atomic. http://java.sun.com/j2se/1.5.0/docs/api/java/util/concu summary.html

Interlocked.CompareExchange method [2].

At the hardware level, X86 has the CMPXCHG (for compare and exchange instruction) while SPARC calls its compare and swap instruction CAS.

Load-linked and store-conditional

The second style of synchronization primitive is a pair of instructions: load-linked (LL) and store-conditional (SC). The idea is that LL reads from an address, followed by a write (SC) to that same address. This SC will complete the store only if the value at the address has not changed since the earlier LL. A number of architectures support such a pair of instructions in cluding the Alpha AXP (`ldl_l`,`stl_c`), IBM PowerPC (`lwarx/stwcx`), and ARM (`ldrex/strex`).

Linearizability

To this point we will have considered parallel programs correct if they were equivalent to their underlying sequential specification.

We now define a different correctness condition for parallel programs with no single sequential specification. *Linearizability* is a correctness condition for such parallel programs. We define linearizable using the following terms:

- A *history* is a sequence of method invocations and responses on a concurrent object by a set of threads.

- A *sequential history* is a history in which each invocation is followed immediately by the corresponding response.

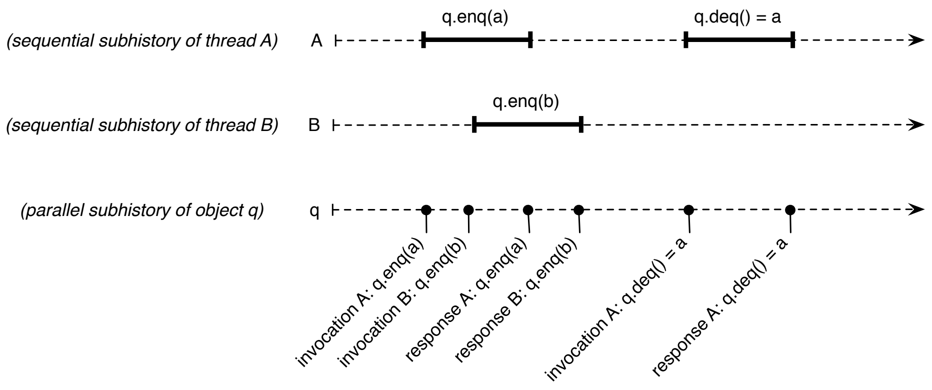
  Consider the histories shown below. There are two sequential subhistories of the two threads, A and B. The third history is the parallel history of the queue object `q`.

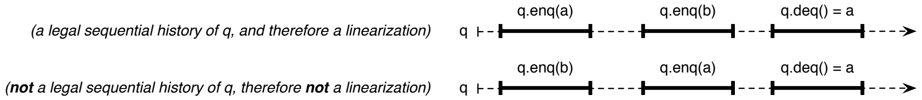
- A *linearization* of a parallel history is a sequential history that is the result of reordering the invocations in the parallel history such that:

  1. If one method call precedes another in the parallel history, the order is preserved in the sequential
  2. If two method calls overlap, the ordering is ambiguous and the calls can be ordered in any convenient way
  3. The resulting sequential history is legal

  Both of the following sequential histories satisfy the first and second requirements, however only the first history satisfies the third, correctness, requirement.

[2] Microsoft Corporation. Interlocked.compareexchange method. http://msdn.microsoft.com/en-us/library/system.threading.interlocked.compareexchang




q.enq(a)
q.enq(b)
q.deq() = a
(a legal sequential history of q, and therefore a linearization)
q
q.enq(b)
q.enq(a)
q.deq() = a
(**not** a legal sequential history of q, therefore **not** a linearization)
q

- An object is *linearizable* if there exists a linearization for all possible parallel histories.

- An alternative, equivalent, definition of linearizability is that an object is linearizable if each method call appears to take effect instantaneously at some moment between its invocation and response.

- It is possible to prove linearizability by finding a *linearization point* for each method, the moment at which the method's effects become visible to the other threads.

Atomic updates of shared state will assist us in identifying linearization points and thus to reason about the linearizability of the concurrent objects.[3]

Non-blocking data structures

Motivation

Processes can be delayed for any number of reasons including OS scheduling preemption, a page fault, or a cache miss. Blocking algorithms suffer significant performance degradation when a processes is halted or delayed at an inopportune time. Non-blocking algorithms are more robust in the face of these events.

[3] Linearizability is discussed at some length and rigorously in "The Art of Multiprocessor Programming" .

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kauffman, first edition, 2008

Locks, as a synchronization mechanism, have many undesirable properties including *priority inversion* when a lower-priority thread is interrupted while holding a lock that a higher priority-thread is trying to acquire (resulting in the delay of a high-priority thread), *convoying* when any thread holding a lock is interrupted resulting in all other threads that are attempting to acquire that lock queueing up and waiting, and worst of all *deadlock* in which threads attempt to lock the same objects in different orders resulting in neither thread being able to proceed.

We are therefore interested in defining and exploring a class of algorithms that do not suffer the above pitfalls. This class of algorithms is called *non-blocking* and an algorithm's membership is satisfied by each of the non-blocking properties defined here.

Execution Intervals

When reasoning about progress properties we will reason about progress in terms of execution intervals. An execution interval is the period of time it takes a thread to carry out some work. Executing a method call, or a critical section, or even some part of a method call are all examples of execution intervals. An execution interval can be *infinite* (it may never terminate), *finite* (it will terminate), or *bounded* (it will terminate within a bounded amount of time.

	some thread is finite	*all threads are finite*	*all threads are bounded*
lock-free	●		
wait-free	●	●	
bounded wait-free	●	●	●

Figure 1: Comparison of non-blocking progress conditions

Both *wait-freedom* and *lock-freedom* are both properties that satisfy the non-blocking progress condition.

A method is wait-free if every call to it finishes in a finite number of steps. An object is wait-free if all of its methods are wait-free.

By contrast, a less strict non-blocking progress condition, *lock-freedom* will allow individual threads to starve, but guarantees system-wide progress. All wait-free algorithms are also lock-free, but not vice versa.

Example: lock-free queue

To understand how such a property is acheived in practice, let us examine an implementation of a lock-free FIFO queue shown in Figure 2. In addition to its constructor, this class has two methods: `enq()` and `deq()`. The code uses atomic `compareAndSet()` operations, as defined earlier in Section , to control updates to shared state. The queue consists of a linked list in which the first node is a dummy

```
public class LockFreeQueue {
    private static type Data = String;

    private static class Node {
        var data:Data = null;
        var next:AtomicReference[Node] = AtomicReference.newAtomicReference[Node](null);
        public def this(data:Data, next:Node) {
            this.data = data;
            this.next = AtomicReference.newAtomicReference[Node](next);
        }
    }

    private var head:AtomicReference[Node];
    private var tail:AtomicReference[Node];

    public def this() {
        val sentinel = new Node(null,null);
        head = AtomicReference.newAtomicReference[Node](sentinel);
        tail = AtomicReference.newAtomicReference[Node](sentinel);
    }

    public def enq(data:Data) {
        var d:Node = new Node(data, null);
        var t:Node = null;
        var n:Node = null;
        do {
            t = tail.get();
            n = t.next.get();
            if (tail.get() != t) continue;
            if (n != null) {
                tail.compareAndSet(t,n);
                continue;
            }
            if (t.next.compareAndSet(null,d)) break;
        } while (true);
        tail.compareAndSet(t,d);
    }

    public def deq() {
        var d:Data=null;
        var h:Node=null;
        var t:Node=null;
        var n:Node=null;
        do {
            h = head.get();
            t = tail.get();
            n = h.next.get();
            if (head.get() != h) continue;
            if (n == null)
                throw new Exception("Nothing␣to␣dequeue!");
            if (t == h)
                tail.compareAndSet(t,n);
            else
                if (head.compareAndSet(h,n)) break;
        } while (true);
        d = n.data;
        n.data = null;
        h.next = null;
        return data;
    }
}
```

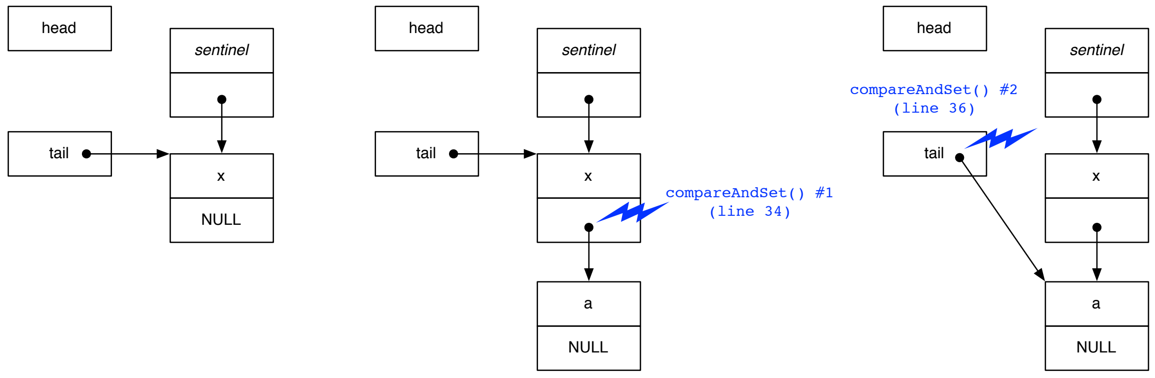
Figure 2: `LockFreeQueue.x10`: X10 implementation of a lock-free queue from Herlihy and Shavit's "Art of Multiprocessor Programming"

node or sentinel. Figure 3 illustrates the standard operation of the `enq()` and `deq()` methods.

The `enq()` method is prepared to encounter two different scenarios in the shared state.

The first scenario (`n == NULL`, line 30) is if the queue is in a clean state where all of the head and tail pointers are correct. `enq()`'s operations in this case are illustrated in the two atomic steps in Figure 3. First it atomically updates the pointer of the `next` pointer of the last element in the queue to point to the new element to be added (line 34). It then atomically updates the `tail` pointer to point to the new tail of the queue (line 36). Because these two operations are not executed atomically, it is possible that another thread will come upon a queue in which the first CAS has occurred, but not the second.

Figure 3: Clean `enq()` operation



The queue starts in a clean state

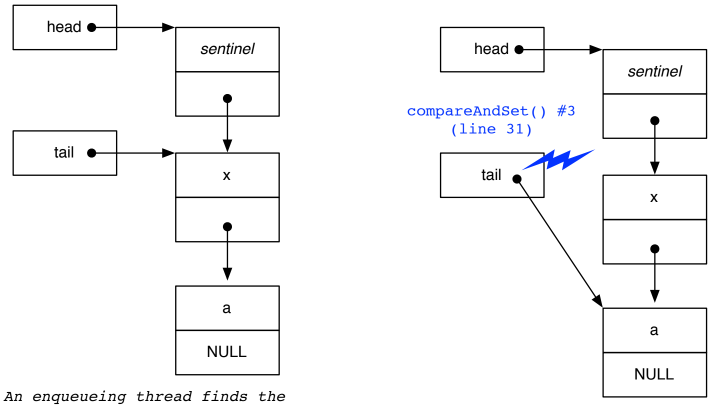
The enqueueing thread first links the new node into the chain

It then updates the tail pointer to point to the new node (which is now the tail)

Implementations of `enq()` must also, therefore, be prepared to encounter and to clean up after a half-completed `enq()` operation as illustrated in Figure 4. This second scenario occurs if (`n != NULL`, line 30). Should thread B call `enq()` only to discover that thread A has stopped halfway through its own `enq()`, thread B will find that the `tail` pointer has not yet been updated. It will then then execute the `compareAndSet()` operation on line 31 to update the `tail` pointer for thread A's `enq()` before returning to the top of the while loop to re-attempt the enqueue of its own new node.

Similarly the `deq()` method is prepared to encounter two states. In the first scenario (if `t != h` on line 51) the dequeueing thread can proceed with its normal dequeue operation as shown below Figure 5.

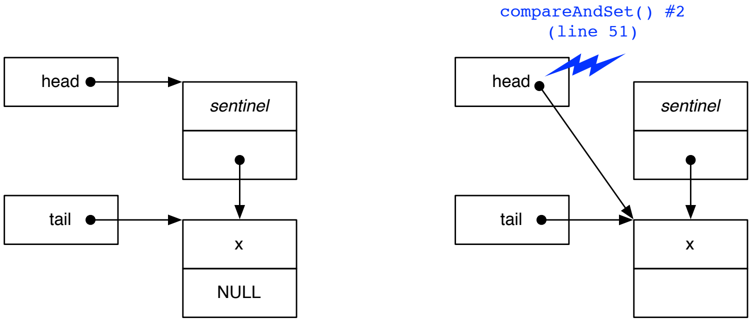
In the second scenario, if the queue is non-empty and `h == t`, the `deq()` method finishes up an in-progress `enq()` operation. While nor-



An enqueueing thread finds the queue in an unclean state (where the tail pointer does not point to the last item in the queue)

Upon recognizing this, the enqueuer first updates the tail pointer completing the incomplete enq() and returning the queue to a clean state

Figure 4: `enq()` encountering another enqueue in progress



The queue starts in a clean state

The dequeuer first reads the first value (x) and then updates advances the head pointer (leaving the previous sentinel to be garbage collected)

Figure 5: Normal `deq()` operation.

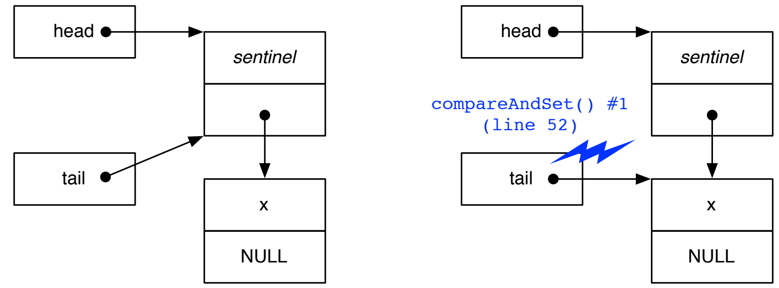
mally enqueues and dequeues are independent, when `h == t`, the tail of the queue is close enough to the head that an incomplete enqueue operation *will* affect the proper operation of the dequeue. In this scenario, illustrated in Figure 6, the `deq()` method needs to advance the tail pointer on line 52 (just like `enq()` does when it finds `tail` lagging) so that `deq()` can later advance `head` without advancing it *past* `tail`.

For further examination, a Java implementation of a similar lock-free queue is discussed in great detail in "The Art of Multiprocessor Programming" [4], Chapter 10.

Implementing locks

A good locking algorithm must have the following three correctness and progress properties. Properties are stated here for two threads,

[4] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kauffman, first edition, 2008



A dequeuer finds the queue in an unclean state where h==t (ln 51) but the queue is not empty (n != null, line 49).

This occurs when some other thread was interrupted halfway through enqueueing an item on an empty list.

The dequeuer() updates the tail pointer so that it can later safely advance the head pointer without causing the tail pointer to lag behind the head pointer.

Figure 6: `deq()` finishing up in-progress enqueue

but can be readily generalized.

- **Mutual exclusion:** The execution of the critical sections by different threads do not overlap in time.
- **Deadlock-freedom (lock-freedom):** Some thread can always proceed. (I.e., both threads will not both be trying and unable to acquire the lock.)
- **Starvation-freedom (wait-freedom):** Every thread that attempts to acquire the lock will *eventually* succeed.

Peterson's lock

We will prove these three properties for Peterson's two-thread lock implementation shown in Figure 7. For further discussion of this code, as well as the Bakery code presented later, refer to "The Art of Multiprocessor Programming" [5], Chapter 2.

Proof of mutual exclusion We will prove mutual exclusion by contradiction.

- Let us assume that execution of the critical section by thread A (CS_A) overlaps with thread B's execution of the critical section (CS_B). In order for each thread to enter the critical section, each thread will have executed the series of operations shown in black in Figure **??**. (The arrows indicate the precedence ordering of the operations.)
- Let us assume, without loss of generality, that thread A was the last to write to `victim` field. This allows us to add the blue arrow

[5] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kauffman, first edition, 2008

```
class Peterson {
    // thread-local index, 0 or 1
    private var flag:Array[Boolean]{self.rank==1};
    private var victim:Int;

    public def this() {
        flag = new Array[Boolean](2,false);
        victim = 0;
    }

    public def lock(i:Int) {
        assert(i ==  0 | i == 1);
        val j:Int = 1 - i;
        flag(i) = true;                  // I'm interested
        victim = i;                      // you go first
        while (flag(j) && victim == i) {}; // wait
  }

    public def unlock(i:Int) {
        flag(i) = false;
    }
}
```

Figure 7: `Peterson.x10`: X10 implementation of a Peterson lock

to Figure **??** and to determine that thread A's subsequent read of `victim` produced the value `A`.

- Since thread A entered the critical section, we know that it succeeded in acquiring the lock. In order for A to acquire the lock, it must have satisfied the condition `victim != A` or `flag(B) == false`. Since we know that `victim == A`, we can infer that `flag(B)` must have been `false`. (Indicated in green.)

- However, we know the last writer to `flag[B]` was thread B, and that it set `flag(B) = true`. This write of `flag(B) = false` followed by a read of `flag(B)` that produces true, indicated by the red arrow, is a contradiction.

Therefore, it is not possible that threads A and B executed the critical section at the same time, and thus this lock implements mutual exclusion.

Proof of starvation-freedom We will prove starvation-freedom of Peterson's lock, also by contradiction.

- Suppose that Peterson's lock is not starvation-free, that some thread, say A, runs forever in the lock() method.

- For a thread to run forever in the lock() method, it must be executing the while loop, waiting for either `flag(B) == false` or `victim == B`.

- What is the other thread, thread B, doing during this time? It must be doing one of three things:
  - CASE 1: B does not want to execute the critical section. In that case, `flag(B)` will be `false` which should cause thread A to break out of the while loop, contradicting our assumptions.
  - CASE 2: Thread B is repeatedly executing the critical section, entering and leaving it repeatedly. Upon entry, thread B sets `victim = B`. The value of the victim field will never change after this because thread A is stuck in the while loop. However it is impossible for thread A to spin while `victim == B`, so we have another contradiction.
  - CASE 3: Thread B may also be stuck in the while loop. For thread B to be stuck in its while loop, `flag(A) == true` and `victim == B`. However, once again, thread A cannot be spinning when `victim == B`, so it is impossible that both threads are stuck in the while loop at the same time.

With a contradiction in all possible cases, we have shown that it is impossible for a thread to execute forever, never completing a call to lock().

Proof of deadlock-freedom Since starvation-freedom is a stronger progress constraint than deadlock freedom, proving starvation-freedom automatically proves deadlock-freedom.

The bakery algorithm

We now examine a lock implementation that improves upon Peterson's algorithm in two ways: it supports more than two threads and it enforces a form of fairness (in this case, first come, first served) amongst threads with respect to the order in which they are granted access to the critical section. The starvation freedom property of Peterson's lock guarantees that every thread that calls lock() will eventually enter the critical section. However, there is nothing in the code to enforce fair access to the critical sections regardless of the scheduling of the threads.

Peterson's has no way of tracking the order in which the threads attempt to acquire lock(). The Bakery algorithm is shown in Figure 9. It is so-named because it is inspired by bakeries in which customers take numbered tickets upon entry and are then served in the order of ticket numbers, using this system to introduce inherent fairness into the lock.

The Bakery algorithm splits the lock acquisition into two sections: the *doorway* and the *waiting* section. The doorway interval (denoted

D_i for thread i) consists of a bounded number of steps. The waiting interval (denoted W_i) is finite but unbounded. The Bakery algorithm enforces the following first come, first-served policy: if $D_A \longrightarrow D_B$ then $CS_A \longrightarrow CS_B$. In other words, if thread A completes its doorway interval before thread B, thread B cannot overtake thread A.

Examining the code for the Bakery algorithm in Figure 9, we see a distributed version of the number-dispensing machine:

- `flag(A)` is a boolean indicating whether or not thread A wants to enter the critical section.
- `label(A)` is an integer indicating A's relative order of entering the bakery.

The labels are compared using the « operator where `(label(i),i) « (label(j),j)` if and only if `label(i) < label(j) || (label(i) == label(j) && i < j)`. In English, this comparator first compares the labels of two threads. If they are equal – which can occur as there is nothing to keep two threads from simultaneously examining the label array, finding the same maximum value, and basing their labels on that – the comparator breaks the tie by using the thread IDs.

It is easy to inspect this code and see that, because threads never reset their labels, that the labels increase monotonically. We will now prove the three base locking properties (implementation of mutual exclustion, deadlock-freedom, starvation-freedom) as well as fairness (FCFS).

Proof of mutual exclusion We will again prove that the Bakery algorithm satisfies mutual exclusion via contradiction. Suppose the opposite: that the critical section of one thread, CS_A, overlaps with the execution some other thread's critical section, CS_B. Suppose also, without loss of generality, that A entered the critical section before B, that `(label(A),A) « (label(B),B)`. For B to have entered its critical section (for CS_B to begin it must have read that `flag(A) == false` (i.e., that A did not want to enter the critical section) or `(label(B),B) « (label(A),A)` (that B had precedence over A). This latter condition contradicts our assumption, so it must be the case that B entered the critical section because it observed `flag(A) == false`. However, by our assumptions A is in the critical section (with `flag(A) == true`) when B enters. Therefore it is impossible that B could have observed `flag(A) == false`. Because the overlapping execution of critical sections leads to a contradiction, we have proof that this algorithm implements mutual exclusion.

Proof of deadlock-freedom The Bakery lock is deadlock free because some waiting thread A must have the uniquely least value of `(label(A),A)`.

The algorithm ensures that that thread will never wait for another thread, and can thus proceed, guaranteeing that the system is free of deadlock.

Proof of first-come, first-served property We will prove first-come, first-served by proving that if A's doorway precedes B's, then A's label is smaller. Inspect the detail of the doorway executions in Figure 8. We see that thread A's setting of its own label precedes B's reading of it and subsequent setting of its own label. This sequence of operations (indicated by the red arrows) implies that `label(A) < label(B)`. So, the only way that B can proceed to the critical section is if `flag(A) == false`. Once A has executed its doorway, this can only happen after A has called unlock(). Thus, the critical section of A must complete before B can begin its own critical section.

Proof of starvation freedom Having proven deadlock-freedom and first-come, first-served, the proof of starvation-freedom comes automatically. Deadlock freedom tells us that one thread can always advance, and first-come, first-served tells us that all threads that want access to the critical section will eventually get it, thus we have proven that all threads can advance and that none will starve.

References

Microsoft Corporation. Interlocked.compareexchange method. http://msdn.microsoft.com/en-us/library/system.threading.interlocked.compareexchange(VS.71).aspx.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kauffman, first edition, 2008.

Sun Microsystems. Package java.util.concurrent.atomic. http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/atomic/package-summary.html.

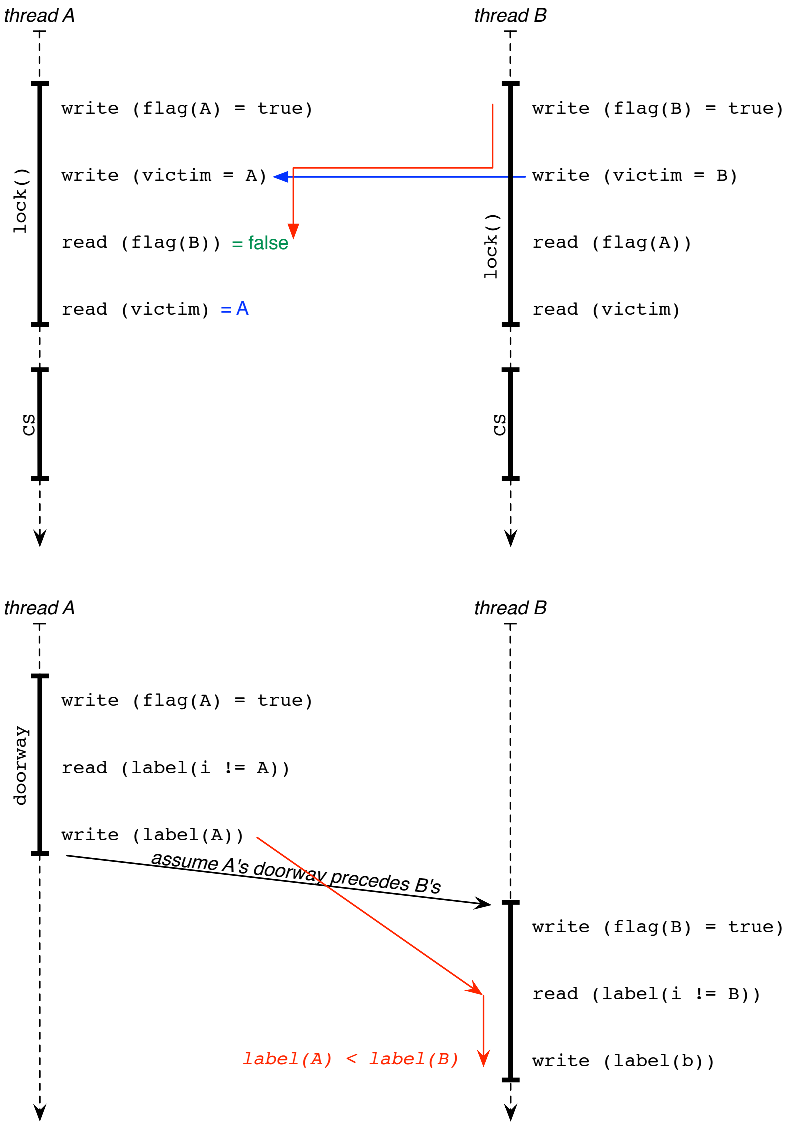


Figure 8: Illustration of Bakery's enforcement of first-come, first-served property

```
class Bakery {
    private var flag:Array[Boolean]{self.rank==1};
    private var label:Array[Long]{self.rank==1};

    public def this(n:Int) {
        flag = new Array[Boolean](n,false);
        label = new Array[Long](n,0L);
    }

    private def nextLabel() {
        var max:Long = 0;
        for (var i:Int = 0; i < label.size; i++) {
            if (label(i) > max) max = label(i);
        }
        return max+1;
    }

    private def someoneElseFirst(i:Int) {
        for (var k:Int = 0; k < label.size; k++) {
            if (flag(k) &&
                ((label(k) < label(i)) || ((label(k) == label(i)) && (k < i))))
                return true;
        }
        return false;
    }

    public def lock(i:Int) {
        // doorway section
        flag(i) = true;
        label(i) = nextLabel();
        // waiting section
        while (someoneElseFirst(i)) {}
    }

    public def unlock(i:Int) {
        flag(i) = false;
    }
}
```

Figure 9: `Bakery.x10`: X10 implementation of a Bakery lock