6.891 Multiprocessor Synchronization — September 27, 1995

Lecturer: Nir Shavit

Handout 3

1 First Homework Assignment

This assignment is due in three weeks.

1. Programmers at the Flaky Computer Corporation designed the following protocol for n-process mutual exclusion with deadlock-freedom.

Shared variables: x, y. Initially $y = 0$.

Code for p_i:

```
** Remainder Region **

try_i
L :
x := i
if y ≠ 0 then goto L
y := 1
x := i
if x ≠ i then goto L
crit_i

** Critical Region **

exit_i
y := 0
rem_i
```

Does this protocol satisfy the two claimed properties? Sketch why it does, or show that it doesn't. (If you quote an impossibility result to say that it doesn't satisfy the properties, then you must still go further and actually exhibit executions in which the properties are violated.)

2. The ℓ-exclusion problem is a variant of the lockout-free mutual exclusion problem designed for a shared memory setting in which some processes might fail by stopping without warning. An implementation must satisfy the following conditions:

ℓ-Exclusion: At any time at most ℓ processes are in their critical regions.

ℓ-Lockout-Freedom: In any execution of the system (a fair one), if strictly fewer than ℓ processes fail (are delayed forever), then any non-failing process that leaves the non-critical section later re-enters it.

Modify Peterson's n-process mutual exclusion algorithm to make it into an ℓ-exclusion algorithm. (Note: The lockout-freedom condition only requires progress if there are fewer than ℓ failures, because strengthening it to allow, say, ℓ failures would lead to the following difficulty. Suppose ℓ processes fail while in the critical region. Then (even though the users eventually return the resources), as far as the rest of the system can tell, they remain in the critical region, which prevents any other processes from entering the critical region.)

3. The following are the definitions of regular and atomic registers:

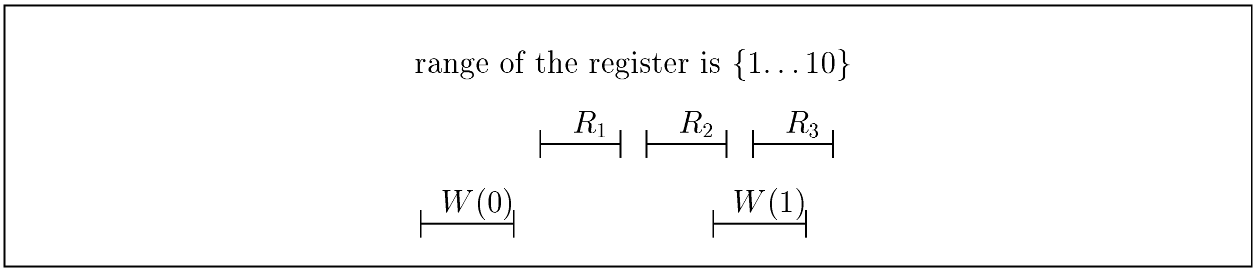

Figure 1: a run of *SRSW Register*

Definition A register is called *Regular* if it has the following properties:

(a) Every *READ* command, which is not concurrent with a *WRITE* command, obtains the last written value.

(b) Every *READ* command which is concurrent with a *WRITE* command obtains either the old or the new value.

For an implementation of Figure **??**, running on *SRSW Regular Register*, the read operations return :

- $R1$ returns the value written before: 0.
- $R2$ and $R3$ return either 0 or 1.

Definition An *Atomic Register* is :

(a) *Regular*.

(b) *Linearizable* to a sequential register.

For an implementation of Figure **??**, running on *SRSW Atomic Register*, the read operations returns :

- $R1$ returns the value written before: 0.
- If $R2$ returns 1 then $R3$ will also return 1.
- If $R2$ returns 0 then $R3$ could return 0 or 1.

```
head, tail, buffer[0 ... n−1] : shared registers

procedure ENQ(x : item) ;
    t := tail ;
    if head − t < n     /* buffer not full */
      then
            buffer[head mod n] := x ;
            head := head + 1 ;
      fi ;
end ENQ ;

procedure DEQ(x : item) ;
    h := head ;
    if h − tail > 0     /* buffer not empty */
      then
            x := buffer[tail mod n] ;
            tail := tail + 1 ;
      fi ;
end DEQ ;
```

Figure 2: *FIFO Queue* algorithm with Read and Write

Given these definitions, what is the weakest type of register (regular or atomic) that can be used for for implementing each of the variables $Q[i]$ and $TURN[j]$ in Peterson's mutual exclusion algorithm (shown in class)? What about the x_i and ℓ_i in Lamport's bakery algorithm? Which (if any) of the two algorithms' correctness properties (mutual-exclusion, no-deadlock, no lockout) would not hold if all the registers were regular?

4. Figure **??** describes a FIFO-queue for one enqueuer and one dequeuer (actually a producer-consumer algorithm). Prove that this is a wait-free implementation of a one-enq-one-deq fifo-queue, by showing that it is linearizable to an object whose sequential specification meets the FIFO rule.

5. The sequential time-stamp system of Israeli-Li that we saw in class, has a range of $O(3^n)$ different label values for n processes (each possible label value is a node in the T^n graph). Devise a sequential time-stamp system that uses less than $O(3^n)$ values (hint – try for $O(n2^n)$ labels).

2 Some Definitions for Today's Lecture

A FIFO queue is a concurrent data structure which provides each of n processors operations of the following types:
Enq(x) - inserting value x to queue.
Deq(x) - removing the value at the top of the queue and returning it in x.

We will expand these definition to fit our concurrent system:
q E(x) A - an Enq event of value x on queue q by process A.
q D(x) A - a Deq event of value x on queue q by process A.

Intuitively, we can define the FIFO nature of the queue by:

FIFO : E(x) $\rightarrow$ E(y) $\Rightarrow$ D(x) $\rightarrow$ D(y)

(meaning that if x is enqueued before y, then it is also dequeued before y).
Our queue is implemented as follows:

```
q : queue = record
        items : array[1...] of item;
        back  : index;
    end record;
```

items - an infinite array data items.
back - the index of the next unoccupied position in *items*.

The implementation of Enq and Deq is shown in Figure ??.

Definition A history H is *linearizable* if it can be extended (by appending zero or more response events) to some history H' such that:
L1 : complete(H') is equivalent to some legal sequential history S, and
L2 : $\longrightarrow_H \;\subseteq\; \longrightarrow_S$.

Informally, extending H to H' captures the idea that some pending invocations may have taken effect even though their responses have not yet been returned to the caller. Extending H to H' restricting attention to complete(H') makes it possible to finish unfinished operations (as shown in Figure 4.1b) or just ignoring them. *L2* states that this apparent sequential interleaving respects the precedence ordering of operations.

3 Bibliography

- *TUXEDO System Release 4.0 – Product Overview.* AT&T, 1990.

```
ENQ(q : queue, x : item);
    mutex_entry;
    i := INC(q.back);
    WRITE(q.items[i], x);
    mutex_exit;
    end ENQ


DEQ(q : queue) returns(item);
    mutex_entry;
    while true do
            range := READ(q.back) − 1;
            for i in  1..range do
                  x := SWAP(q.items[i], null);
                  if x ≠ null then return (x)
                     end if
                  end do ;
            end do ;
    mutex_exit;
    end DEQ


INC(x : item);
    saved_x := x;
    x := x + 1;
    return(saved_x);
    end INC


SWAP(x, y : item);
    saved_x := x;
    x := y;
    return(saved_x);
    end SWAP
```

Figure 3: Code for ENQ and DEQ

- J. Burns and N. Lynch. Bounds on Shared Memory for Mutual Exclusion *Information and Computation*, 107(2):171-184, December 1993.
- A. Israeli and M. Li. Bounded time stamps. *Distributed Computing*, (6), 1993.
- M.P Herlihy and J.M. Wing, Linearizability: A correctness condition for concurrent objects, ACM Trans. on Programming Languages and Systems, Vol.1,No. 3, July 1990, pp.463-492.