Lecturer: Nir Shavit

Handout 11-s

The following are the answers to the first homework assignment.

Question 1

Part 1

The given algorithm does not guarantee progress. Consider the following execution:

1	2
$x = 1$	
y initially 0	
$y = 1$	
$x = 1$	
	$x = 2$
	$y \neq 0$ so goto L
$x \neq 1$, so goto L	

$y = 1$ and can never be reset because both processes will fail on the test of $y \neq 0$.

Part 2

The algorithm does not provide mutual exclusion.

1	2
$x = 1$	
	$x = 2$
y initially 0	
	y initially 0
$y = 1$	
	$y = 1$
$x = 1$	
$crit_1$	
	$x = 2$
	$crit_2$

Mutual exclusion is violated by allowing both processes execute their critical sections at the same time.

Question 2

procedure l-exclusion for process i:

```
remainder section;
for j := 1 to n − l
    Q[i] := j;
    TURN[j] := i;
    repeat until ((TURN[j] <> i) or (there are at least n-l processes with Q[k] < j));
    end for ;
crit_i;
Q[i] := 0;
```

Consider the following modification to Peterson's algorithm:

```
Initially Q[1..n] = 0
for j := 1 to n − l
    Q[i] := j;
    TURN[j] := i;
    repeat until (|{k ≠ i|Q[k] < j}| ≥ n − l or TURN[j] ≠ i);
    end for ;
crit_i;
Q[i] := 0;
```

By modifying two properties of Peterson's algorithm, we can adapt it for l-exclusion. We first allow l processes into the critical region instead of just one by decreasing the bound for j. Under maximum contention, exactly one process waits at each level, so having $n-l$ levels allows l processes into the critical region.

The second property is more subtle. In its original form, if one process is in the critical region and another enters the trying section, Peterson's algorithm blocks the second process at the first level. The $\forall k \neq i(Q[k] < 1)$ test will fail, and if this is the only process in the trying section $TURN[1] \neq i$ will also fail because this process just set $TURN$ to its id. The problem with this property in the l-exclusion case is that if one process dies in the critical region, no other process will be able to enter (assuming no other processes enter the trying region - once they do the $TURN$ variable will change and the waiting process will progress one level).

To ensure that l processes can always enter the critical section, the test is modified to ensure that at lease $n-l$ processes are at a lower level than the currently trying process. If there are $n-l$ below it, then there can be at most l (including itself) either in the critical or at this or higher level in the trying region. If there are fewer than l above it, the process can progress up a level.

$TURN$ is still necessary to ensure progress. Assume two processes are at level 1, $l-1$ processes are in the critical region and $n-l-1$ processes are at level 0. Both processes would fail the at least $n-l$ processes below me test, and so remain at level 1. If there are $l-1$ failures, no more progress would be possible, violating l-exclusion. The $TURN$ variable allows one of the processors to advance a level at which point the process is in the critical region, or the $n-l$ processes below me test will succeed, and the process will progress into the critical region.

Question 3

In this question people had a problem only with Lamport's bakery algorithm.

The variable l can be a regular register, and let us prove that the algorithm satisfies mutual exclusion.

Assume that two processes P_i and P_j are in the critical section. Let P_i be the one having the smaller l, at the last repeat loop of P_j (before it entered the critical section) and that P_j found $(l_j, j) < (l_i, i)$ or $x_i = false$.

If it found $(l_j, j) < (l_i, i)$, because l_i is a regular register, P_j read the current value of l_i (which is smaller than l_j) or previous value which is smaller than the current value (l value is increasing), a contradiction.

If it found $x_i = false$, this means that P_j read x_j while P_i was before the doorway phase or at the doorway phase before setting l_i (and while P_j was in the repeat loop), a contradiction to the assumption that $l_i > l_j$.

Also it is easy to prove that when using a regular register as the l variable the algorithm is deadlock free, lockout free, and satisfies the FCFS propoerty.

Regular registers are sufficient for all of the variables in Peterson's algorithm and in Lamport's bakery algorithm.

Part 1

In Peterson's algorithm, both Q and $TURN$ are sampled repeatedly in a loop. Therefore, the algorithm only necessitates that the registers eventually return a stable value, so regular registers are sufficient.

The purpose of $TURN$ in Peterson's algorithm is to ensure that at most 1 process inhabits a level which is under contention. Therefore it is only important that the last process (call it i) to enter level j, and hence the last process to set $TURN[j]$, reads $TURN[j] = i$. Since the read is done after the write for the same process, regularity is sufficient. Any transients can only delay another process from entering the next level, it can not prevent it.

The purpose of Q in Peterson's algorithm is to reflect the number of processes below a certain level. If a process i reads an old value $Q[k] \geq j$ while it is being written, either k is still at a higher or equal level than i in which case it is correct for the test to fail, or that process just executed $Q[k] := 0$ which i will eventually register the next time it tests $Q[k]$. If a process i reads an old value which is smaller than j while j is being written, it will advance a level, but the algorithm ensures that the process that just wrote $Q[k]$ will stay at level j because $TURN[j]$ will equal k.

Part 2

In Lamport's bakery algorithm, l_i can return its current or previous value because ties are resolved by process identifier.

x_i in Lamport's bakery algorithm is similar to $TURN$ in Peterson's algorithm. It only guarantees progress. Assume all x_i are *false*. If a regular register returns the old value of *false* for some x_k, then a waiting process (call it i) may enter the critical section because the first part of the test (all x_j's are *false*) succeeds. The algorithm guarantees that $l_k > l_i$ so k will not also enter the critical region. If not all x_j are *false*, then the test will fail independent of the value of x_k.

Question 4

The most natural way to linearize this implementation is to pick the point at which the operation is said to happen as the point at which either *head* or *tail* is incremented. Of course, if the operations fail, then this never happens. In this case, we pick the point at which the operation first read *head* or *tail* as the time at which the operation happens. This assumes that reads and writes to *head*, *tail* are atomic.

First, we should note that these operations have no loops, so this implementation must be wait-free!

Second, we describe the FIFO spec as saying that an enq should always succeed unless the queue is full, and a deq should always succeed unless the queue is empty. What we need to show is that these implementations succeed and fail in the right places, and that they reutrn the right values.

In the following cases, when I say that an enq or deq operation "fails", what I mean is that it skips past the body of the if statement and leaves *head*, *tail* unchanged.

Case 1 The queue is empty ($head == tail$)

Case 1a The enq precedes the deq

Since the deq never changes *head* or *tail* before its point of linearizability, enq will read these correctly. Since the queue is empty, it will succeed, and after it happens, $head = tail + 1$. If we assume that the deq fails, then it must read $head \leq tail$, but since this is the after the enq, we know that $head = tail + 1$. Since this is a contradiction, deq must succeed. This is correct, according to the spec.

Case 2 The queue is neither empty nor full. ($0 < head - tail < n$)

Case 2a The enq precedes the deq

By reasoning similar to above, the deq can't change *head* or *tail* before enq reads them. Thus, enq will succeed. At any time after the enq, *head* is 1 bigger than it used to be. Thus, since $0 < head - tail < head + 1 - tail$, deq will also succeed. This meets the spec.

Case 3 The queue is full ($head - tail == n$)

Case 3a The enq precedes the deq

Since the enq is first, it must read *head*, *tail* s.t. $head - tail = n$ and will therefore fail. Since it fails, it does not change *head* or *tail*, and the deq will read *head*, *tail* s.t. $head - tail > 0$, thus succeeding. This is a correct behavior.

Case 3b The deq precedes the enq

Since the deq is first, it must read *head*, *tail* s.t. $head - tail = n > 0$, and will therefore succeed. After it happens, $head - (tail + 1) < n$. If we assume that the enq fails, then it must read the incremented version of *tail*, since the point at which a failing enq is linearized to, is the point where it reads *tail*. Since at the beginning, $head - tail == n$,

and it read the incremented *tail*, it must succeed. This is a contradiction, so our assumption must be false. Both ops succeed, which meets the spec.

These cases do not include the situation where $0 > head - tail > n$, since it is easy to see from the above cases that when $head - tail = 0$, the deq operation fails (so the queue can't get any smaller) and when $head - tail = n$ the enq operation fails (so the queue can't get any bigger).

This shows that in all cases these ops correctly either succeed or fail. We also have to show that the right results are returned. This is easy, if we observe that *head* and *tail* increase monotonically, and that for each value of *head*, there is exactly one entry written into the buffer, and for each corresponding value of *tail*, this same value is returned. The fact that entries are written into the buffer mod n is irrelevant since at all times $0 \leq head - tail \leq n$, which means that enq never overwrites anything until deq gets it, and vice versa.

Question 5

Let each label (i, s) consist of the process number i and a string of bits n long, where n is the number of processes. Let $s[k]$ denote the kth bit of the string s. The ordering between two labels (i, s) and (j, t) is determined as:

$(i, s) > (j, t)$ iff $((i > j$ && xor($s[j]$,$t[i]$)) || $(i < j$ && !xor($s[j]$, $t[i]$))

where && denotes logical and

|| denotes logical or

xor denotes logical exclusive or

! denotes logical not

There are $n2^n$ possible labels.

The time stamp system need only remember which labels each process has. It can then construct a new label for a process i, by constructing a string of bits s such that for any process j with label (j, t)

if $i > j$ then $s[j] := !t[i]$
if $i < j$ then $s[j] := t[i]$
if $i = j$ then $s[i] :=$ anything

and returning (i, s)

Note that for the label (i, s), $s[i]$ is irrelevant, since a label never needs to be compared against itself. Hence, you could actually remove that bit from the string and pack the rest in a little tighter, which would reduce your number of possible labels to $n2^{n-1}$.