Lecturer: Nir Shavit

Handout 8

1 Administrative

Today it's time for you guys to start scribing... The scribes should:

1. Write everything during class
2. Ask if they don't understand something
3. Put it into LaTeX format
4. Include all explanations given in class

The final exam will be handed out December 6, and will be due in class on December 13. The problems will be similar to those on the problem set, just a little harder.

Class will now start at 10:10, *sharp!*

2 Review from last lecture

In lecture 7 we described the *consensus problem* and started showing how we can use it to derive a hierarchy of concurrent objects such that no object at one level can implement in a *wait-free* fashion any object at higher levels. The basic idea was the following: each object has an associated *consensus number*, which is the maximum number of processes for which the object can solve the consensus problem. We showed that it is impossible to construct a wait-free implementation of an object with consensus number n from any object with a lower consensus number. We continued to prove that *atomic registers* have consensus number 1, and that registers with *read-modify-write* (RMW) operations have consensus number of at least 2. In fact we showed that (RMW) operations that apply functions from an interfering set f have consensus number of exactly 2. We also talked about *Compare&Swap* and *queues* but did not include them in the handout. In the current lecture we continue to prove lemmas concerning some consensus numbers associated with different objects and we conclude by showing that any object with consensus number of at least n can implement any other object in a system of n or less processes.

We have also seen two other techniques for determining the consensus number for an object. By demonstrating a reduction, such as in figure 1, it is possible to set an upper bound on the consensus number. By giving an implementation of consensus using an object, it is possible to set a lower bound on the consensus number.

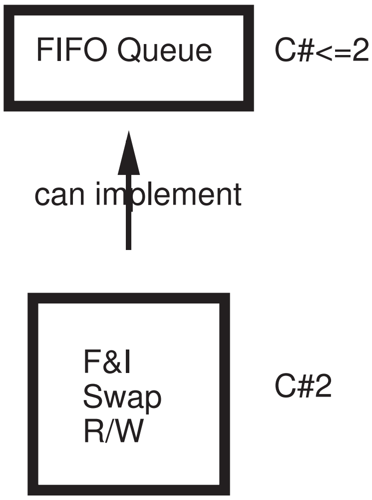

Figure 1: Proving a maximum consensus number via implementation

3 Queues

The non-blocking implementation of a FIFO queue, that we saw, wasn't a wait-free implementation (Critical sections were used.). This raises the question : *What is the consensus number of a FIFO queue ?*

Lemma .1 *FIFO queue has a consensus number of at least 2.*

Proof: We will give a protocol for two processes P and Q, using a FIFO queue q. (See Figure

```
DECIDE(input : value)returns(value);
  begin
  prefer[P]:=input;
  if DEQ(q)=0
    then return (prefer[P])
    else  return (prefer[Q])
    end if
  end DECIDE
```

Figure 2: Decide for P (Decide for Q is symmetric).

2). The queue q is initialized by performing on it *ENQ(0)* followed by *ENQ(1)*. Its head contains 0 and its tail contains 1. Every process has its own register in the array *PREFER*. The process that will perform the first *DEQ* will get the zero and will return his input. The other process will get one, when performing his *DEQ* and so returns the input of the first process. ∎

Lemma .2 *The consensus number of a FIFO queue is not greater than 2.*

Proof: The proof is by contradiction. Assume we have a consensus protocol for three processes P,Q and R. As before, we maneuver the protocol to a state s where P and Q are each about to make a decision step. Assume that P's operation would carry the protocol to an X-valent state and Q's to a Y-valent state. The rest is a case analysis.

First, suppose P and Q both execute *deq* operations. Let *s'* be the protocol state if P dequeues and then Q dequeues, and let *s"* be the protocol state if the dequeues occur in the opposite order. Since *s'* is X-valent, there exist a history fragment from *s'*, consisting entirely of operations of R, yielding decision value X. But *s'* and *s"* differ only in the internal states of P and Q; thus the protocol has the same history fragment from *s"*, a contradiction because *s"* is Y-valent. In other words, R has no way to know what was the order of the *dequeue* operations, and this is in contradiction of them being a decision step for the protocol.

Second, suppose P performs an *enq* and Q a *deq*. If the queue is nonempty, the contradiction is immediate because the two operations commute. One should keep in mind that this is a queue, i.e, in a *deq* operation the first item in the queue is returned while in the *enq* operation the item is added to the end of the queue. If the queue isn't empty the same values are returned no matter what the order was. Thus, if the queue isn't empty, even P and Q can not tell what the order between their operations was, they'll get the same results in the two cases. R won't be able to know the order of the operations of P and Q. If the queue is empty, the Y-valent state (*s"*) reached if Q dequeues and then P enqueues is indistinguishable to R from the X-valent state (*s'*) reached if P alone enqueues . (We allow a dequeue operation on an empty queue. If its illegal to do this then such order of operations is not allowed and we shouldn't concern ourselves with it.)

Finally, suppose both P and Q perform *enq* operations. Look first at an empty queue (at state *s*):

- P and Q enqueue items *p* and *q* in that order. (Once P is the first to enqueue, and in the other run Q enqueue first.)
- Run P until it dequeues. (Since the only way to observe the queue's state is via the *deq* operation, P cannot decide before it observes one of *p* or *q*.)
- Run Q until it dequeues.

At state *s'* the queue's head is *p* and the queue's tail is *q*. At state *s"* the queue's head is *q* and the queue's tail is *p*. The left history is X-valent, and the right history is Y-valent. But at state *s"* the queue is empty in both histories, so R can't know what to decide.

If the queue wasn't empty at state *s* then at states *s'*, *s"* we will have to run P until it dequeues the *p* it enqueued after state *s*, or until it dequeues the *q* that Q enqueued.(P can't decide until it observes *p* or *q*, because until then the head of the queue is the same in both histories.). ■

Corollary .3 *A FIFO queue has a consensus number 2.*

4 Compare&Swap

Compare&Swap is a primitive that has an infinite consensus number. It is described in Figure 3.

```
Compare&Swap(r : register, old : value, new : value) returns value
    begin
    previous:=r;
    if previous=old then r:=new
       end if
    return (previous);
    end Compare&Swap
```

Figure 3: Compare&Swap

Theorem .4 *Compare&Swap has an infinite consensus number.*

Proof: In Figure 4 is given a consensus protocol.

The Compare&Swap register is initialized to $\perp$($\perp$ can't be an input value for the regis-

```
DECIDE(input : value) returns value
    begin
    first:=Compare&Swap(r,⊥,input);
    if first=⊥
       then return (input)
       else  return (first)
       end if
    end DECIDE
```

Figure 4: Compare&Swap : n-process consensus

ter).The first process to perform Compare&Swap on r is the only one that will see the $\perp$ and so will decide on his input. The other processes will not see a $\perp$, they will see the input of the first process, and so will decide on his input value and won't make the swap. ∎

5 Universality Results

An object is *universal* if it implements any other object. We are going to prove that any object with consensus number n is universal in a system of n (or fewer) processes (figure 5). Any object we could wish to implement can be viewed as a non-deterministic finite state machine, or more specifically an I/O automaton. It has a domain of invocations INVOC, a domain of results RESULT, and a domain of states STATE. An object's behavior can be specified by the following relation *apply*:

$$apply \subset INVOC \times STATE \times STATE \times RESULT.$$

This specification means that if $< p,\ s,\ s',\ r > \in$ *apply* then applying operation p in state s leaves the object in a state s' and returns result r. *apply* is a relation (rather than a function) because the operation may be non deterministic.

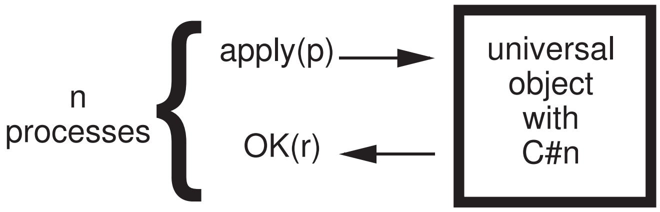

Figure 5: How a universal object can be used

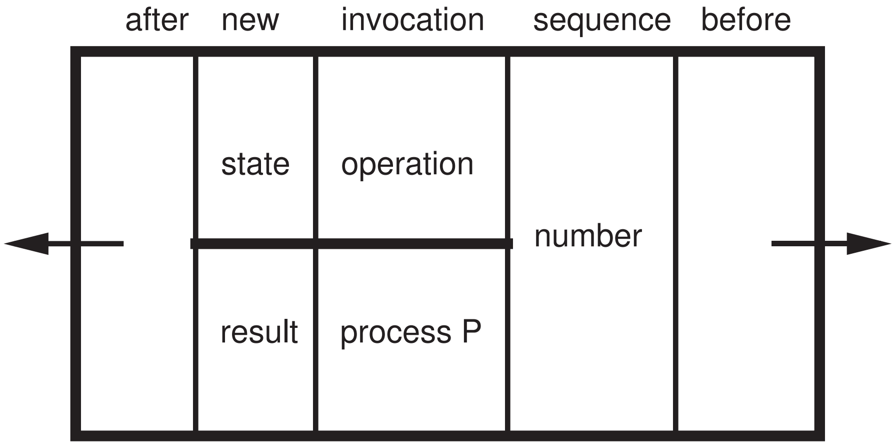

Figure 6: Operation structure used in the Universality Algorithm

In our construction an object is represented by a linked list where the sequence of cells represents the sequence of operations applied to the object (and hence the object sequence of states). Each cell contains the operation performed, it's result and the states of the object before and after the execution of the operation (Figure 6). We assume cells can hold sequence numbers of unbounded size. The idea is that each process that wants to apply an operation to the shared object tries to thread a cell to the end of the list, and we will guarantee that it will succeed to do so in a finite number of steps. The main problems we have to solve are:

- The competition between processes that wish to perform an operation on the shared object in the same time.
- If a process had stopped, for some reason, in the middle of performing an operation on the shared object, another process should complete the operation or else this won't be a wait-free implementation.

Two shared arrays used by the protocol, besides the linked list are ANNOUNCE[1..n] - where each cell ANNOUNCE[Q] is a pointer to the cell process Q is currently trying to thread, and HEAD[1..n] - where each cell HEAD[Q] is a pointer to the last cell in the list process identified (figure 7).

The protocol is given in Figure 8, and before we prove it's correctness we give an informal description of a run:

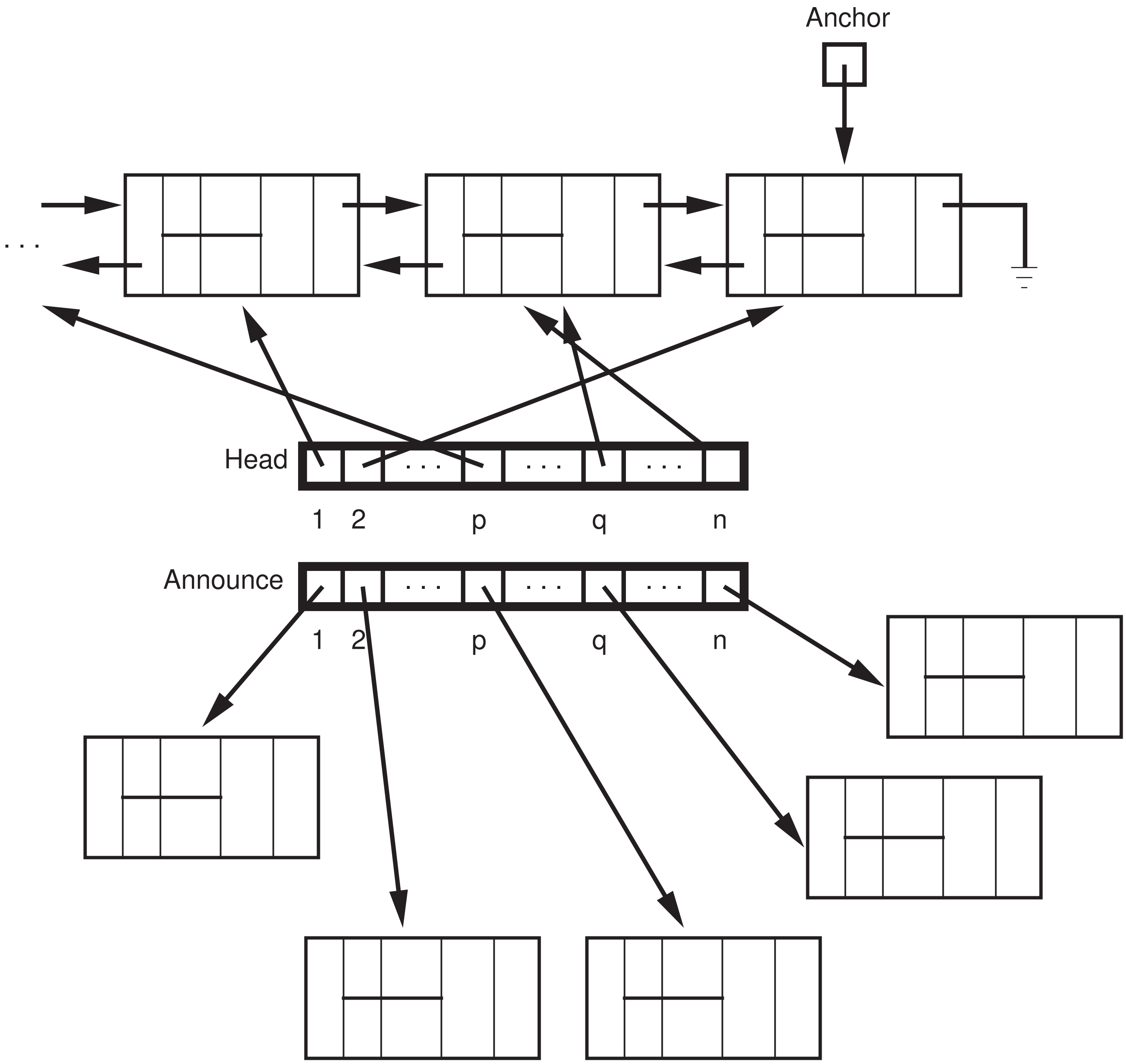

Figure 7: Data structures used in Universal Object algorithm

A process P that wants to perform an operation on the shared object, first chooses a new cell, initializes its sequence number to 0, and writes in it the operation it wants to perform. It also puts in the array ANNOUNCE a pointer to its cell. Every process checks what is the last cell that was added to the list, this is done by scanning the HEAD array.

As long as the cell of a certain process was not added to the list, it does the following things:

1. It writes to C the cell with maximal sequence number in HEAD.

```
UNIVERSAL(what : invoc)returns(result);
    begin
    mine: cell := [ seq: 0,
     inv: what,
     new: create(consensus-object),
     after:null ]
    ⟨ ANNOUNCE[P] := mine; start(P):=max(HEAD)⟩
    for Q do
        HEAD[P] := max(HEAD[P], HEAD[Q])
        end for
    while ANNOUNCE[P].seq = 0 do
            c: ↑ cell := HEAD[P]
            help : ↑ cell := ANNOUNCE[(c.seq mod n) + 1]
            if help.seq = 0 then prefer := help
              else               prefer := ANNOUNCE[P]
              end if
            d := decide (c.after,prefer)
            decide(d.new,apply(d.inv,c.new.state))
            d.seq := c.seq + 1
            ⟨HEAD[P] := d ; (∀ Q) concur(Q):=concur(Q)∪ {d}⟩
            end while
    ⟨HEAD[P] := ANNOUNCE[P]; (∀ Q) concur(Q):=concur(Q)∪ {ANNOUNCE[P]} ⟩
    return (ANNOUNCE[P].new.result)
    end UNIVERSAL
```

Figure 8: A Universal construction

2. It tries to help another process that is currently trying to thread a new cell:
   - If there is such a process whose sequential number is exactly (c.seq mod n) + 1, it will choose to help it, otherwise
   - It will choose to help himself.
3. The next cell to be added to the list is 'elected' between the processes through the use of a consensus object.
4. All the processes which are currently trying to add their cells perform (in a consensus object) the operation of the cell that was chosen to be added.
5. The process then changes the max(HEAD) to the cell that it helped to add, and thus the process moves along the ANNOUNCE array.

Note that the before pointer is not really needed in the record objects since the algorithm never traverses backwards.

Theorem .5 *The protocol given correctly implements any object.*

Proof: The object is linearizable almost by definition, since unique sequence numbers are attached to each object.

Next we need to show that the protocol is wait-free. For example, *how do we know that a process will not just keep helping and helping?*

Next we will prove that after n helps, the process's operation will be done, i.e. this algorithm is wait-free. This proof is done with the help of a couple ghost variables:

- concur(P) = the set of cells whose addresses have been stored in the head array since P's last announce. These operations can be thought of as being announced "concurrently" with P's announce.
- start(P) = the value of max(head) at P's last announce.

By the definitions of the ghost variables, it follows that $\mid concur(P) \mid + start(P) = max(head)$.

Lemma .6 $\mid concur(P) \mid > n \Rightarrow announce(P) \in head$

Proof: Basically, after n operations are performed collectively by the processes, P's operation is done. Suppose this was not true. Then we have a situation like figure 9. Since Q inserted an object after start(P) was assigned, that object must be in concur(P). And since R came after Q, it had to have seen P's object in announce[P], unless that object had already been inserted. Therefore, P's operation will be done after at most n operations are performed. ■

Lemma .7 $max(head) \geq start(P)$

Proof: Since the numbers in the head array don't decrease, the number to which start(P) is assigned will be in the head array, or will be replaced by a greater number. So max(head) will always be at least start(P). ■

Lemma .8 *The maximum value, or a greater value, is found in the "for Q do" loop*

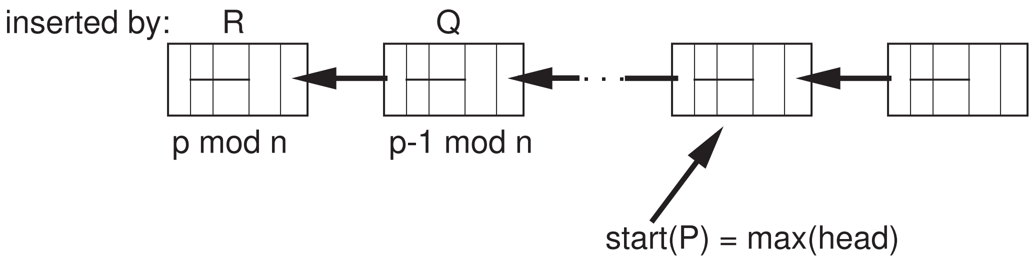

Figure 9: $\mid concur(P) \mid > n$, yet P's operation has not been inserted yet

Proof: When start(P) is assigned, some element in the head array has that value. Let i be the index of that element in the array. When the for loop gets to element i, either the maximum value will be there, or a greater value will be there. This holds since, for each particular head element, the value is non-decreasing. ∎

Lemma .9 *After P's operation is put in the list,* $head[P].seq \geq start(P)$

Proof: It is not possible that P's new operation is in the operation list until announce[P] is set. start(P) is set pricisely when announce[P] is set. Therefore, P's new operation can be inserted into the list only after start(P) is assigned, and thus this new operation cannot get a lower sequence number than start(P). ∎

Lemma .10 *After P's operation is inserted,* $\mid concur(P) \mid \geq head[P].seq - start(P) \geq 0$

Proof: The second inequality follows from Lemma .9. As stated earlier, $\mid concur(P) \mid + start(P) = max(head)$, and the sequence number for P's operation is equal to max(head) when it is inserted. After the operation is inserted, others may also be inserted and also added to concur(P). So $\mid concur(P) \mid$ may continue to increase, and we have $\mid concur(P) \mid + start(P) \geq head[P].seq$. The first inequality follows directly from this equation. ∎

Since we know the difference $head[P].seq - start(P)$ begins at 0, and that the difference can increase by 1 at most n times before P's operation is inserted, we have an upper bound on the number of helps that P will perform and thus this algorithm is wait-free. ∎

An important aspect of this algorithm is that the processes first try to help others and then help themselves, thus assuring global progress. If processes first helped themselves, then it is possible that a particular process would always fail and get "stuck".

6 Dealing with real machines

We now start a new topic, dealing with the real world of machines and their behavior.

We have shown that different primitive operations have different powers, and that we have universal objects which can be used to implement any object wait-free. *So why don't we do everything wait-free?* Firstly, doing things on the granularity of the machine isn't enough. We want to use granularity that optimizes the effectiveness of the object. This really increases the complexity of wait-free objects, when compared to that of MutEx. We will take stronger primitives, build wait-free objects, and see when we do better with locks and when we do better without locks.

The three basic problems that arise when dealing with the complexity of synchronization on actual multiprocessors are:

1. Memory Contention - several processes trying to apply an operation to the same memory module, which in turn can serve only one process at a time.
2. Communication Contention - A delay caused as a result of the serialization of requests moving on the same communication channel.
3. Communication Latency Time - The *worst/average* time it takes a massage to move through the communication network.

7 Bibliography

- M.P. Herlihy, *Wait-free synchronization* , same as last week.