Universal Construction (cont'd)

Reference: [Her91] or [AW98], Section 15.3.

Given objects that can solve (wait free) consensus (and registers) among n processes, construct implementation of object of type T for n procs, where T is any type.

Idea

Store a linked list of operations performed on the simulated object of type T in the order that they are linearized. (Once you know which operations precede the opeation you are simulating, you can figure out the response for your operation.)

- Each node in linked list looks like (Figure1):

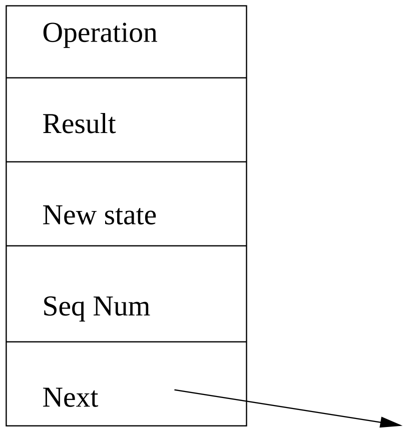

Figure 1: Node in Linked List

- "Next" pointer for each node in list is set by solving consensus.
- To try to set a pointer to P, run Consensus algorithm:

1.) using P as input value
2.) set pointer to output of consensus

Note there is a separate consensus algorithm associated with each node, so two procs trying to update pointers of different nodes are solving separate consensus problems; they don't interfere with one another. For example, if the system provides us with compare&swap objects, recall that we can build a consensus algorithm using a single object (see Feb 6 lecture). Then we would use one compare&swap object to implement each node's pointer.

Each process keeps a pointer to last node it has seen. (Figure2)

- To simulate operation:

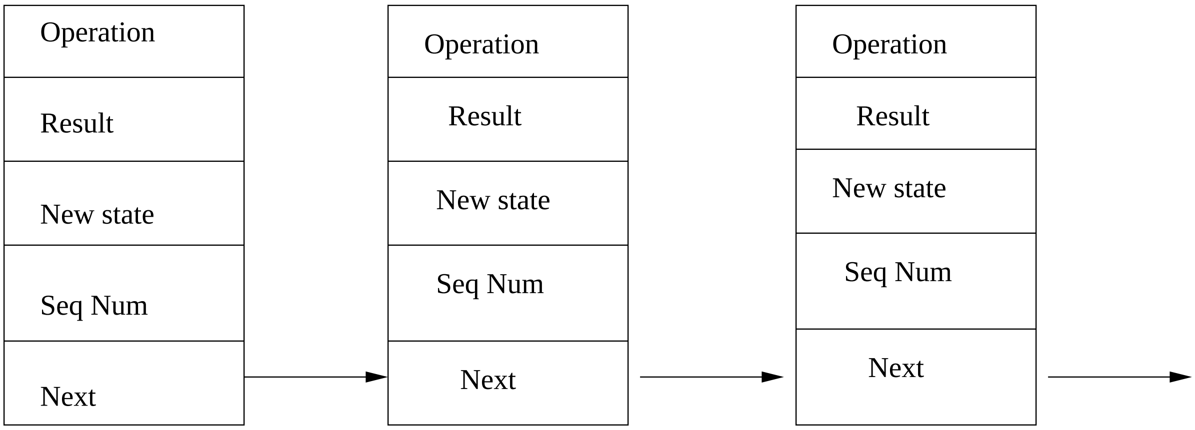

Figure 2: Simulating Operation in Shared Memory

create new node containing the name of the operation.
loop % try to add it to last
    exit when node added to list
    try to set last.next to point to new node (using consensus as described above)
    if you fail, fill in result, new state, seq# fields of the node that did win
    last = last.next
end loop

This gives a non-blocking implementation

means: If some processes are trying to simulate operations, eventually one will succeed.

However, it's not wait-free

wait-free: every operation will eventually succeed.

Reason it's not wait-free:

if $P1$ is slow

$P2$ might keep adding more and more nodes to end of list faster than $P1$ can follow the pointers, so $P1$ never gets to end of list.

key idea to go from non-blocking to wait-free: processes should help one another to add their operations to end of list.

If a process is running slowly, put his operation on the end of the list for him.

This can be done using the following scheme: process i has first priority for position j in the list if $j \bmod n = i$.

This requires processes to know about the operations that others are trying to add to the list. We use an additional array Announce$[1..n]$ in shared memory for this.

Code for Pi:

create new node with your operation name in it and write a pointer to that node in
    Announce(i) (in shared memory) (figure3)
loop
  exit when seq# for your node has been filled in
  (**)
  $k \leftarrow$ seq# of last mod n % process who has priority for next position
  if Announce[k] = null or Announce[k].seq# $\neq$ null (i.e. Pk has no operation that hasn't
      already been added to list) then
    $k \leftarrow i$ % try to add my own operation instead
  % now try to add Pk's operation
  propose Announce[k] to consensus algorithm corresponding to last$_i$.next
  last$_i$.next $\leftarrow$ result of consensus
  fill in seq#, result, new state in successor
  last$_i$ $\leftarrow$ last$_i$.next
end loop
output result in Announce[i].result

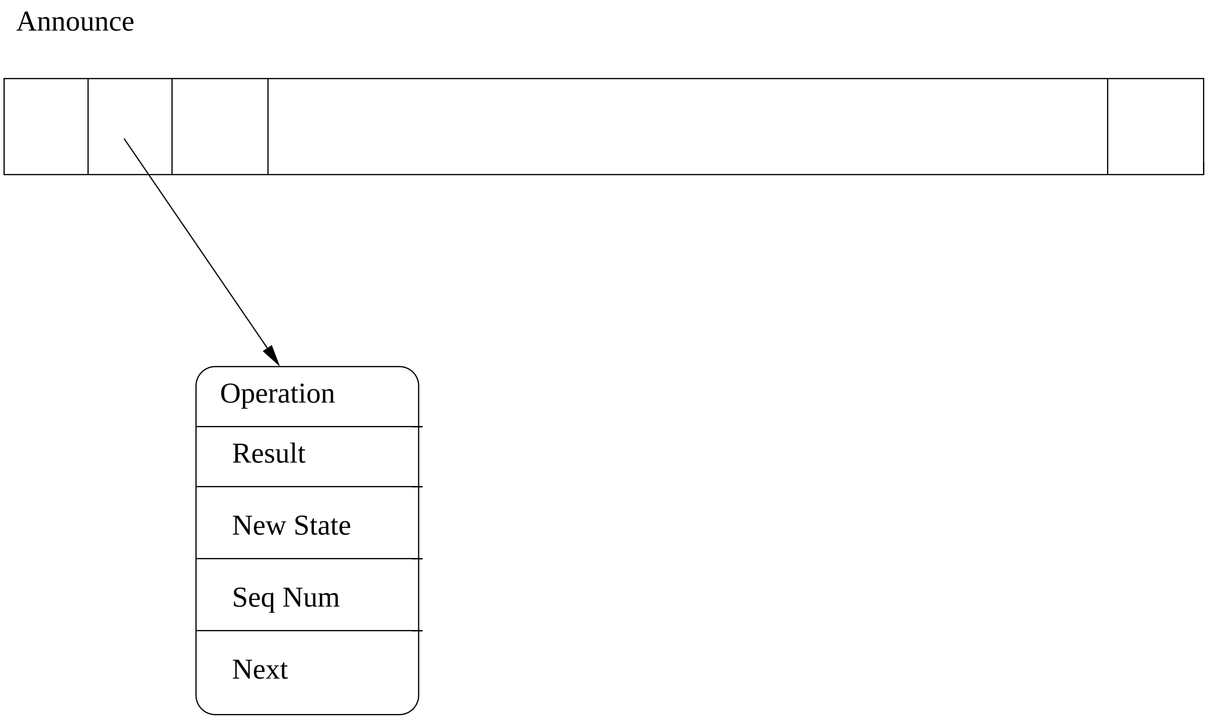

Figure 3: Announce for wait-free simulation

Why is it wait-free?

once your operation has been put in Announce[i] array, each other process can only try to put an operation in one of your slots once (because that other process started its iteration of its loop before Announce[i] was written).

Problem 1: if other processes have done lots of operations since you last took a step, number of iterations you will have to perform to reach end of list be arbitrarily large.

Solution (sketch): Instead of using private variables for last_i, put them in shared memory. To skip to the end of the list more quickly, peek at other processes last pointers. Add the following line at the position marked (**) in the code above: $\text{last}_i \leftarrow \max(\text{last}_1, \ldots, \text{last}_n)$. Now, whenever this line is executed, either (1) last_i will point to the last or 2nd last elt of the list, which means P_i will be able to append its node soon OR (2) some node(s) has been added while P_i was executing the line. If (2) happens $O(n)$ times, then P_i's cell will have been added to the list by somebody. Either way, P_i can halt and return the result of its operation after $O(n)$ iterations, so the total time to simulate any operation is $O(n^2)$.

Problem 2: unbounded space needed

Solution: suffices to keep $O(n^3)$ nodes in list and rellocate memory when nodes aren't needed anymore. (Needs extra bookkeeping.) See Herlihy's paper for details.

There are quite a few other papers on improved universal constructions (eg. Afek et al. in STOC95, Chandra et al. in PODC98, Herlihy in TOPLAS 1993, Jayanti and Toueg in WDAG92) and also papers that prove that there are fundamental limitations on the efficiency of universal constructions (2 papers by Jayanti in PODC98).