Section 7
Wait-Free Simulations of Arbitrary Objects

Wait-Free Simulations of Arbitrary Shared Objects

- The consensus problem cannot be solved using only read/write registers.
- Most modern multiprocessors provide some set of "stronger" hardware primitive for coordination, like LL/SC or Compare&Swap.
- We investigate the following question:

  "Given two types of (linearizable) shared objects, X and Y, is there a wait-free simulation of object type Y using only objects of type X and read/write registers?"

- We will first answer this question for the weaker termination property called non-blocking (or lock-freedom):

  "Lock-freedom states that there is a finite execution fragment starting at any point of an admissible execution in which some high-level operations are pending, at which a process completes one of the pending operations."

- Lock-freedom is a weaker property than wait-freedom which states that eventually all processes should complete their operations.
- Lock-freedom allows starvation to occur!
- The distinction between wait-free and lock-free algorithms is similar to the distinction between no-lockout and no-deadlock algorithms for mutual exclusion.

Example: A FIFO Queue

Algorithm 1: Consensus algorithm for two processors, using a FIFO queue: code for processor p_i, $i = 0, 1$.

Initially $Q = \langle 0 \rangle$ and $Prefer[i] = \perp, i = 0, 1$

```
1: Prefer[i] := x                                   // write your input
2: val := deq(Q)
3: if val = 0 then y := x          // dequeued the first element, decide on your input
4: else y := Prefer[1 − i]                          // decide on other's input
```

- The operations supported by a FIFO queue are:
  - [enq(Q,x), ack(Q)],
  - [deq(Q), return(Q,x)],
    where x can be any value that can be stored in the queue (deq(Q) returns $\perp$ if the queue is empty).

Theorem 1

- Algorithm 1 solves consensus for two processes.

Example: A FIFO Queue

Theorem 2

- There is no wait-free simulation of a FIFO queue with read/write objects, for any number of processes.

Proof:

- **If there was a wait-free simulation of FIFO queues with read/write objects, then there would be a wait-free consensus algorithm, for two processes, using only read/write objects.**
- **This is a contradiction to the FLP result!!!**

Theorem 3

- There is no n-process, wait-free consensus algorithm using only FIFO queues and read/write objects, if $n \geq 3$.

Proof: Using valence arguments as in previous section.

Left as an exercise!!!

The strong Compare&Swap Primitive!

```
value Compare&Swap(X: memory address; old, new: value) {
    previous = X;
    if (previous == old) then X = new;
    return previous;
}
```

Algorithm 2: Consensus algorithm for any number of processors, using compare&swap: code for processor p_i, $0 \leq i \leq n-1$.

Initially $First = \perp$

```
1:  v := compare&swap(First,⊥,x)
2:  if v = ⊥ then                          // this is the first compare&swap
3:      y := x                                  // decide on your own input
4:  else y := v                          // decide on someone else's input
```

Theorem 4

- Algorithm 2 solves consensus for any number of processes using a single Compare&Swap object.

The Wait-Free Hierarchy

- Atomic objects can be categorized according to a criterion which is based on their ability to support a consensus algorithm for a certain number of processes.
- Object type X solves wait-free n-processes consensus if there exists an asynchronous consensus algorithm for n processes using only shared objects of type X and read/write objects.
- The consensus number of object type X is n, denoted CN(X) = n, if n is the largest value for which X solves wait-free n-processes consensus. The consensus number is infinity if X solves wait-free n-processes consenus for every n.

✓ The consensus number of any object X is at least 1, because any object trivially solves wait-free one-process consensus.

The Wait-Free Hierarchy

- *For each object type X which is the smallest value that CN(X) can have?*

- ❑ The CN of a read/write register is 1.
- ❑ The CN of the following atomic shared objects is 2: test&set, swap, fetch&add, stacks, queues.
- ❑ The CN of a Compare&Swap register is ∞.
- ❑ There exists a hierarchy of object types based on their CN.
- ✓ It has been proved that there are object types with CN = m, for each value of m>0.

The Wait-Free Hierarchy

Theorem 5

- If CN(X) = m and CN(Y) = n > m, then there is no wait-free simulation of Y with X and read/write objects in a system with more than m processes.

Proof: Assume, by the way of contradiction, that there is a wait-free implementation of Y from objects of type X and read/write registers in a system with k > m processes.

- Denote l = min{k,n}. Note that l > m.
- We argue that there exists a wait-free l-processes consensus algorithm using objects of type X and read/write objects.
- Since l ≤ n, there exists a wait-free l-processes consensus algorithm, A, using objects of type Y and read/write objects.
- We can obtain another algorithm A' by replacing each type Y object with a wait-free simulation of it using objects of type X and read/write registers.
- A' is a wait-free l-processes consensus algorithm using objects of type X and read/write objects ⇒ CN(X) ≥ l > m.
  This is a contradiction!

The Wait-Free Hierarchy

Corollary 6

- There is no wait-free simulation of any object with consensus number greater than 1 using read/write objects.

Corollary 7

- There is no wait-free simulation of any object with consensus number greater than 2 using FIFO queues and read/write objects.

Universality

- An object is universal if it, together with read/write objects, wait-free simulates any other object.

We will prove that:

- Any object X whose consensus number is n is universal in a system of at most n processes.

Note: **This does not imply that X is universal in any system with m > n processes!**

Main Ideas

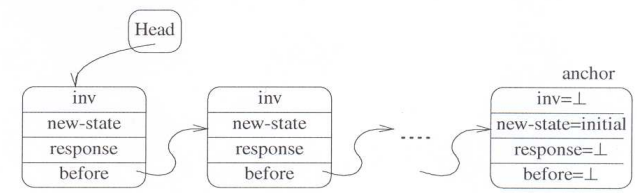
- We present a universal algorithm for wait-free simulating any object in a system of n processes using only n-processes consensus objects and read/write objects.
- An n-processes consensus object `Obj` is a data structure that allows n processes to solve consensus. It provides a single operation `[decide(obj,in), return(Obj, out)]`, where `in` and `out` are taken from some domain of values.
  - The set of operation sequences consists of all sequences of operations in which all `out` values are equal to some `in` value.

A Non-Blocking Universal Construction Using Compare&Swap

Main Ideas

- We represent the object as a shared linked list, which contains the ordered sequence of operations applied to the object.
- To apply an operation to the object, a process has to thread it at the head of the linked list.
- A Compare&Swap object, called Head, is used to manage the head of the list.
- An operation is represented by a shared record of type opr with the following components:
  - inv: the operation invocation including its parameters;
  - new-state: the new state of the object, after applying the operation;
  - response: the response of the operation, including its return value;
  - before: a pointer to a record of the previous operation on the object.
- The initial value of the object is represented by a special anchor record, of type opr, with the new-state field equal to the initial state of the object.

A Non-Blocking Universal Construction Using Compare&Swap



Algorithm 3: A non-blocking universal algorithm using Compare&Swap; code for process p_i, $0 \le i \le n-1$.

Initially Head points to the anchor record;

1. when *inv* occurs:
2. allocate a new opr record pointed to by *point* with *point→inv = inv;*
3. repeat
4. *h := Head;*
5. *point →new-state, point →response* := apply(*inv, h →new-state*);
6. *point →before := h;*
7. until Compare&Swap(*Head, h, point*) = *h;*
8. enable the output indicated by *point →response; // operation response*

A Non-Blocking Universal Construction Using Compare&Swap

Theorem 8

- Algorithm 3 is a non-blocking universal algorithm for n processes.

Proof

- The desired linearization is derived from the ordering of operations in the linked list. So, proving linearizability is straightforward.
- The algorithm is non-blocking.
  - If a process does not succeed in threading its operation in the linked list, it must be that the Compare&Swap operation executed by some other process has threaded its operation in the list.
- The algorithm is not wait-free since the same process might repeatedly succeed to thread its operation, locking all other processes out of access to the shared object.

Disadvantages ☹

- The algorithm uses Compare&Swap instead of consensus objects
- It is not wait-free
- It uses an unbounded amount of space

A Non-Blocking Universal Construction Using Consensus Objects

1st Effort

- Replace the Compare&Swap object with a consensus object.

Problem 1

- A consensus object an be used only once; after the first process wins the consensus and threads its operation, the consensus object will always return the same value.

Solution

- A consensus object is associated with each record of the linked list.
- We replace the before field with a field called after, which is a consensus object pointing to the next operation applied to the object.

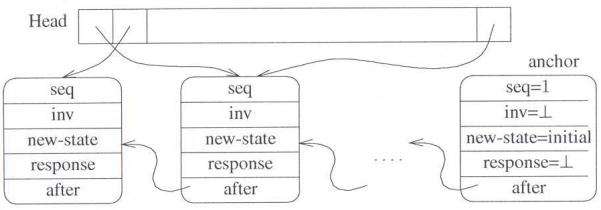
Problem 2

- *How can each process locate the record at the head of the list?*

Solution

- Have each process maintain a pointer to the last record it has seen at the head of the list.
- These pointers are kept in a shared array called Head.
- This information might be stale!
- Sequence numbers are also used so that later operations get higher sequence numbers.

A Non-Blocking Universal Construction Using Consensus Objects



Algorithm 4: A non-blocking universal algorithm using consensus objects; code for process p_i, $0 \le i \le n-1$.

Initially Head[j] points to the anchor record, for all j, $0 \le j \le n-1$;

1. when *inv* occurs: *// operation invocation, including parameters*
2. allocate a new opr record pointed to by *point* with *point→inv = inv;*
3. for j := 0 to n-1 do *// find record with highest sequence number*
4. if (*Head[j]→seq > Head[i] →seq)* then *Head[i] := Head[j];*
5. repeat
6. *win := decide(Head[i] →after, point);* *// try to thread your record*
7. *win →seq = Head[i] →seq + 1;*
8. *win →new-state, win →response* := apply(win→*inv, Head[i]→new-state*);
9. *Head[i] := win;* *// point to the record at the head of the list*
10. until *win = point;*
11. enable the output indicated by *point →response;* *// operation response*

A Non-Blocking Universal Construction Using Consensus Objects

- Showing linearizability is straightforward (in a way similar to the previous algorithm).
- For each configuration C, in an execution a, let:
  max-head(C) = max{Head[i]→seq | $0 \le i \le n-1$}
- For each i, Head[i]→seq is monotonically non-decreasing during a.

Properties of the Algorithm

- The algorithm is non-blocking.
  - If a process p_i performs an unbounded number of steps, then max-head is not bounded. So, other processes succeed in threading their operations to the list.
- The algorithm is not wait-free.

A Wait-Free Universal Construction Using Consensus Objects

- We use the method of helping, according to which each process helps other processes to perform their operations and not being locked out from accessing the data structure.

Problem 1

- How do we know which processes are trying to apply an operation to the object?

Solution

- Keep an additional shared array Announce[0..n-1], the ith entry, Announe[i], of which is a pointer to the record that p_i is currently trying to thread in the list.

Problem 2

- How to choose the process to help in a way that guarantees that this process will succeed in applying its operation?

Solution

- A priority scheme is used, and a priority is given, for each sequence number, to some process that has a pending operation.
- Priority is given in a round-robin way:
  - If p_i has a pending operation, then it has priority in applying the kth operation where k = i mod n.

A Wait-Free Universal Construction Using Consensus Objects

Algorithm 5: A wait-free universal algorithm using consensus objects; code for process pi, 0 ≤ i ≤ n-1.

Initially Head[j] and Announce[j] point to the anchor record, for all j, 0 ≤ j ≤ n-1;

1. when *inv* occurs: *// operation invocation, including parameters*
2. allocate a new opr record pointed to by *Announce[i]*
   with *Announce[i]→inv := inv* and *Announce[i] →seq =0;*
3. for j := 0 to n-1 do *// find record with highest sequence number*
4. if (*Head[j]→seq > Head[i] →seq)* then *Head[i] := Head[j];*
5. while (Announce[i] *→seq = 0*) do
6. priority := (Head[i] *→seq+1) mod n;*
7. *if (Announce[priority] →seq = 0) then point := Announce[priority];*
8. *else point := Announce[i];*
9. *win := decide(Head[i] →after, point);* *// try to thread your record*
10. *win →new-state, win →response* := apply(win*→inv, Head[i]→new-state*);
11. *win →seq := win;* *// point to the record at the head of the list*
12. enable the output indicated by *win →response; // operation response*

Theorem 10

- There exists a wait-free simulation of any object for n processes using only n-processes consensus objects and read/write objects. The step complexity of the algorithm is O(n).

A Wait-Free Universal Construction Using Consensus Objects

Theorem

- There exists a wait-free implementation of any object for n processes, using only n-processes consensus objects and read/write objects. Each process completes any operation within O(n) of its own steps, regardless of the behavior of other processes.

Proof

- Let C_1 be the 1st configuration at which p_i has expressed its interest to execute an operation op_i.
- For each configuration C, max-head(C) is the maximum sequence number of any entry in the Head array. So, max-head(C) continuously increases.
- Let C_2 be the first configuration after C_1 at which it holds that max-head(C_2) mod n = i-1 and let C_3 be the first configuration after C_2 at which it holds that max-head(C_3) mod n = i+1. The operation of process p_i has been inserted in the linked list by C_3.

Theorem

- Any object X with CN(X) = n is universal in a system with at most n processes.

Bounding the Memory Requirements

- There are two types of memory unboundedness in the algorithm:
  - the number of records used to represent an object;
  - the values of the sequence numbers grow linearly, without bound, with the number of operations applied to the simulated object.
- We describe a mechanism to control the first type of unboundedness.

Basic Idea

- Recycle the records used for the representation of the object.
  - Each process maintains a pool of records belonging to it;
  - for each operation, the process takes some free records from its pool;
  - A record can be reclaimed if no process is going to access it.

Difficulty

- Which of the records already threaded on the list will not be accessed anymore and can be recycled?

Bounding the Memory Requirements

- Consider some record r threaded on the list, belonging to process p_i, with sequence number k.
- Let p_j be a process that may access r.
- Then, p_j's record is threaded with sequence number k+n or less.
- The processes that may access r are the processes whose records are threaded as numbers k+1, k+2, …, k+n on the list.
  - Note: These records do not necessarily belong to n different processors but may represent several operations by the same processor.
- We add to opr an array, *released[1..n]* of binary variables.
- Before a record is used, all entries of *released[]* are set to *false*.
- If a record has been threaded as number k on the list, then *released[r] = true* means that the process whose record was threaded as number k+r on the list has completed its operation.
- When a processor's record is threaded as number k', it sets *released[r] = true* in record k'-r, for r = 1,…,n.
- When *released[r] = true* for all *r= 1,…,n,* then the record can be recycled.

Bounding the Memory Requirements

Algorithm 5: A bounded-space, wait-free universal algorithm with using consensus objects; code for process p_i, $0 \le i \le n-1$.

Initially Head[j] and Announce[j] point to the anchor record, for all j, $0 \le j \le n-1$;

```
1.  when inv occurs:                    // operation invocation, including parameters
2.      let point point to a record in Pool such that
            point→released[1] = … = point→released[n] = true
            and set point→inv to inv,
3.      for r:=1 to n do point→released[r] := false;
4.      Announce[i] := point;
5.      for j := 0 to n-1 do                    // find record with highest sequence number
6.          if (Head[j]→seq > Head[i] →seq) then Head[i] := Head[j];
7.      while (Announce[i]→seq = 0) do
8.              priority := (Head[i] →seq+1) mod n;
9.              if (Announce[priority] →seq = 0) then point := Announce[priority];
10.             else point := Announce[i];
11.             win := decide(Head[i] →after, point);     // try to thread your record
12.             win→before := Head;
13.             win →new-state, win →response := apply(win→inv, Head[i]→new-state);
14.             win →seq := Head[i] →seq +1; // point to the record at the head of the list
15.             Head[i] := win;

16.     temp := Announce[i] →before;
17.     for r := 1 to n do                                  //go to n records before
18.         if (temp != anchor) then
19.             before-temp := temp→before;
20.             temp→released[r] := true;                   // release record
21.             temp := before-temp;
22.     enable the output indicated by Announce[i]→response;    // operation response
```

Handling Non-Determinism

- The universal algorithms described so far assumed that operations on the simulated object are *deterministic*.
  - Given the current state of the object and the invocation (the operation to be applied and its parameters), the next state of the object, as well as the return value of the operation, are unique.
  - Example of non-deterministic object: an object representing an unordered set with a choose operation returning an arbitrary element of the set.

Main Ideas on How to Handle Non-Determinism

- If we leave the new-state and response fields of the opr record as read/write objects, it is possible to get inconsistencies as different processes write new (and possibly different) values for the new-state of the response fields.

- Solution
  - We modify the opr record type so that the new state and response value are stored jointly in a single consensus object.
    - We replace the simple writing of new-state and response fields with a decide operation of the consensus object, using as input the local computation of a new state and response (using apply).

Employing Randomized Consensus

Relaxation of Liveness Condition

- The new condition is probabilistic, i.e., it requires operations to terminate only with high probability.
- In this way, randomized wait-free simulations of shared objects are defined.
- Randomized consensus objects can be implemented from read/write registers.
- Thus:
  - there are randomized wait-free simulations of any object from read/write objects, and
  - there is no hierarchy of objects if termination has to be guaranteed only with high probability.