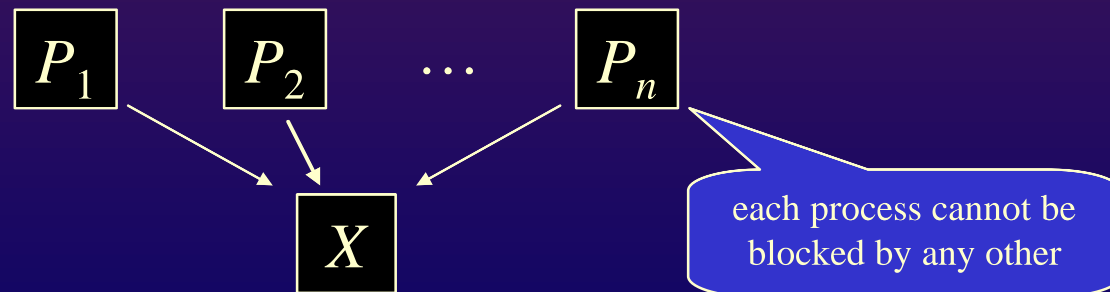
Concurrency Control

Arne John Glenstrup

Motivation

- A concurrent system experiences *partial failures*
- The system should be robust against this
- Thus we need *wait-free* objects for interprocess communication

An object is *wait-free* if any operation on it will complete in a finite number of steps, regardless of how other processes use the object



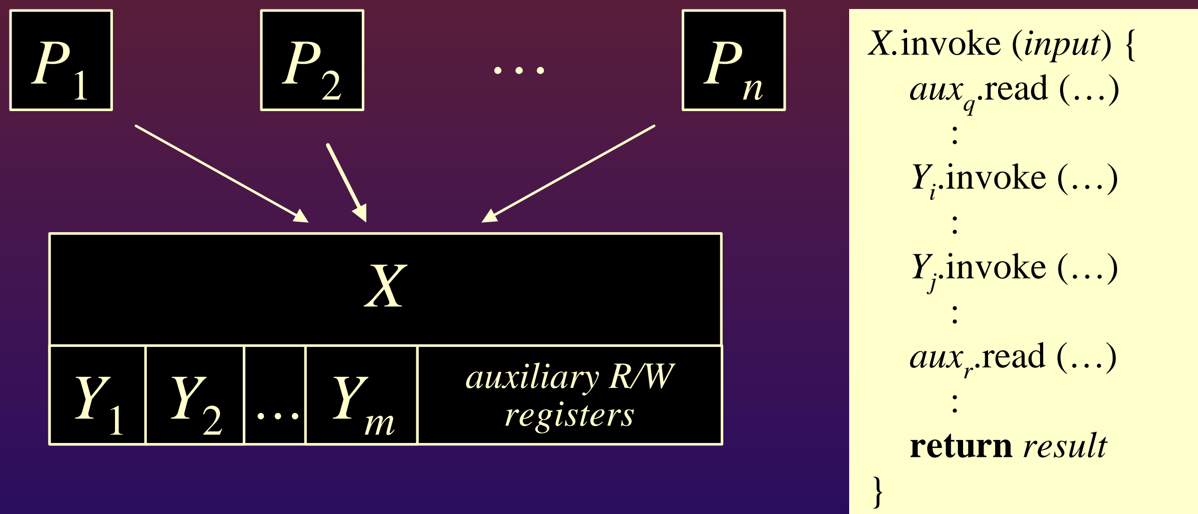
Wait-free or not?

- atomic R/W register — wait-free
- waiting semaphore — non-wait-free
- object accessed using a critical region — non-wait-free
- blocking queue — non-wait-free
- non-blocking queue — wait-free
- stack — wait-free

Building Wait-free Objects

- Key question:

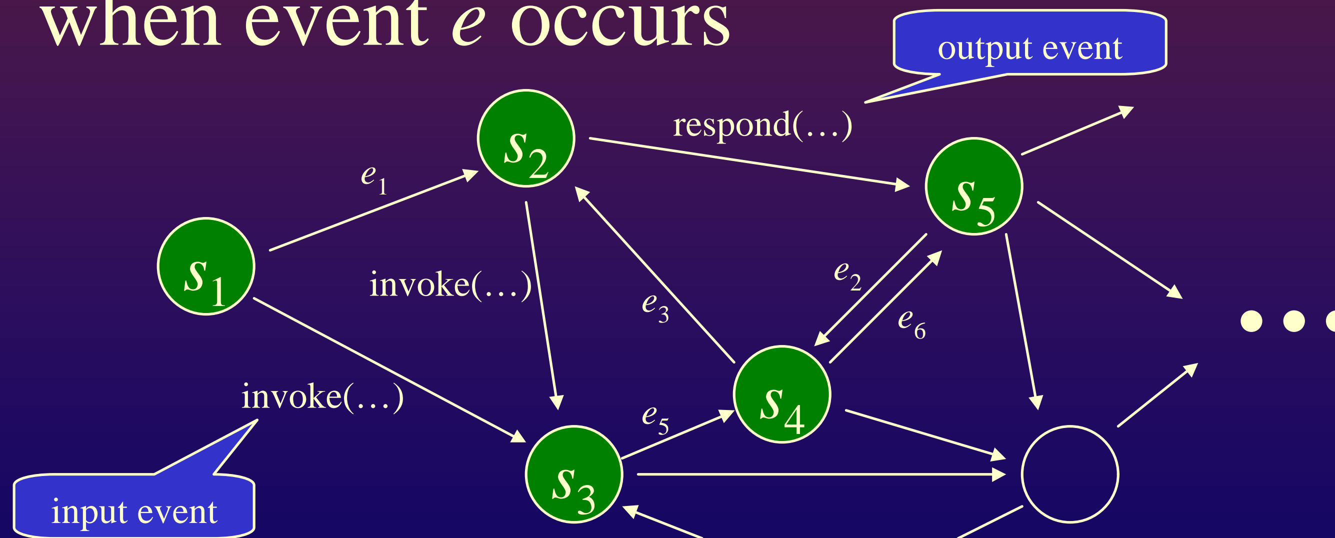
Given wait-free objects *X*, *Y*, how do we determine whether we can build *X* from (some instances of) *Y?*



- Consequence: a hierarchy of objects according to "implementation strength"

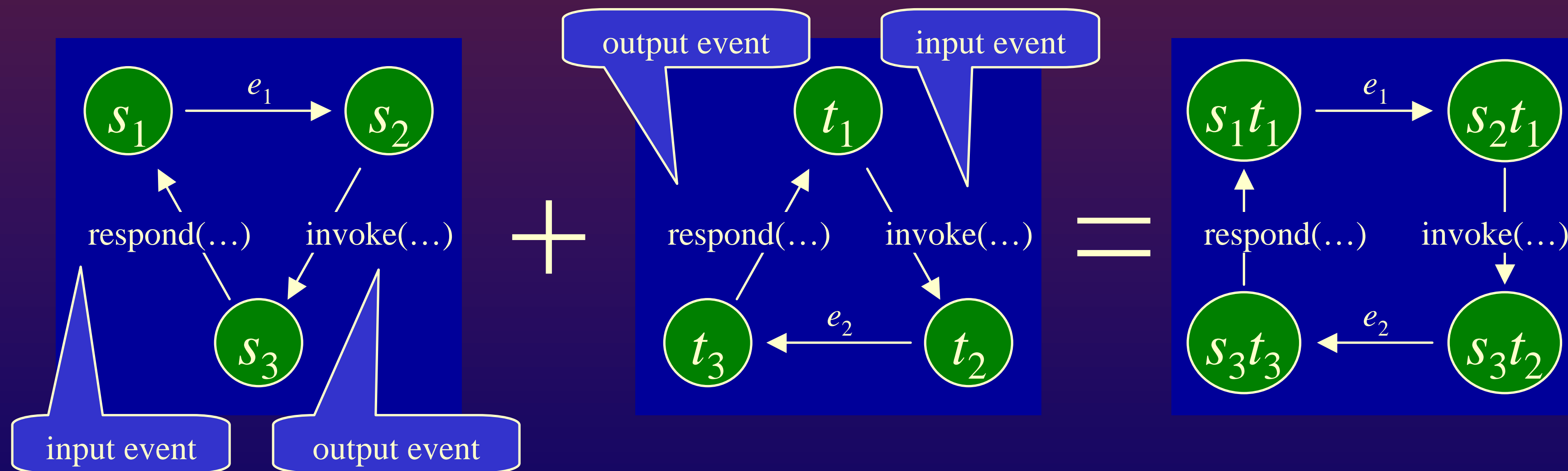
Modeling Processes and Objects

- Processes and objects are modelled by I/O automata
- An I/O automaton is tuple (*states*, *events*, *steps*)
- Events are *input*, *output* or *internal events*
- A *step* (*s´*, *e*, *s*) takes the object from state *s´* to *s* when event *e* occurs



Composing I/O Automata

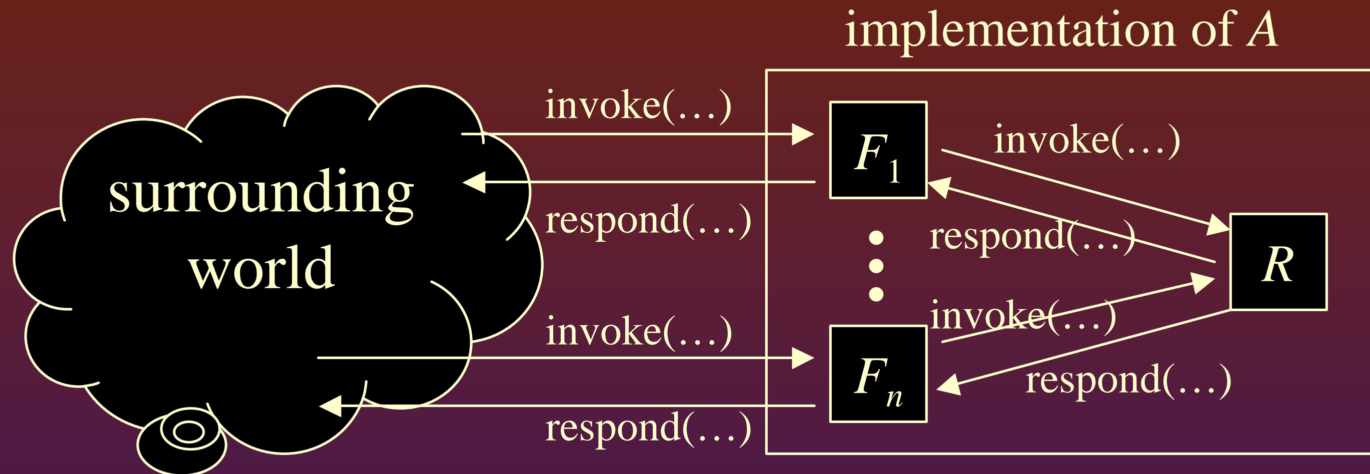
- Automata A_1 and A_2 can be composed if they share no internal or output events
- $states\ (A_1+A_2) = states\ (A_1) \times states\ (A_2)$
- $events(A_1+A_2) = events(A_1)$? $events(A_2)$



Concurrent systems

- An object *A* as well as a process *P* is modelled by an I/O automaton
- Output events for *P* and input events for *A* are ”invoke(*P*, *operation*, *A*)”
- Input events for *P* and output events for *A* are ”respond(*P*, *result*, *A*)”
- A *concurrent system* $\{P_1,\ldots,P_n;A_1,\ldots,A_m\}$ is simply the composition of the I/O automata

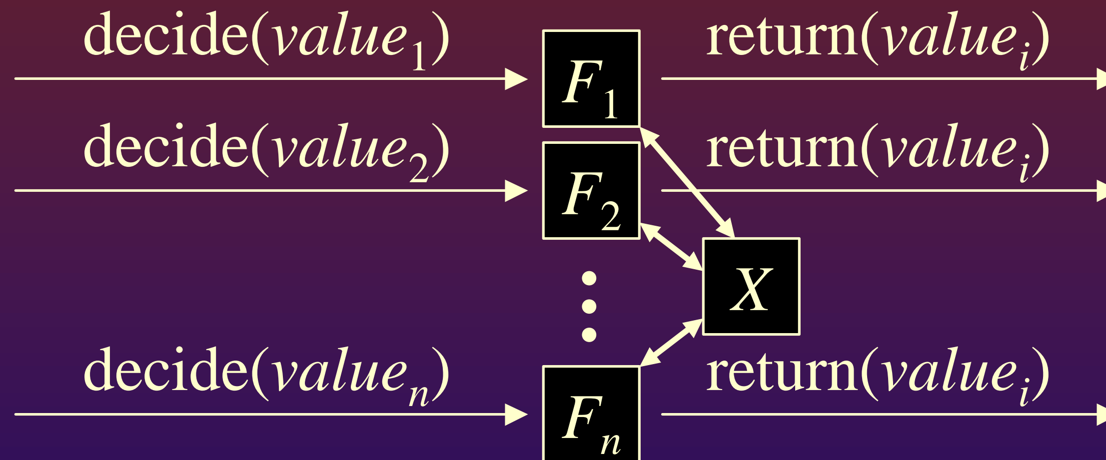
Implementing Objects



- $\{F_1,\ldots,F_n;R\}$ is an *implementation* of object A
- R is the data structure representing A's state
- $F_1,\ldots,F_n$ are *front-end processes* that interact between the surrounding world and R
- Front-ends communicate only via R

Consensus Protocols

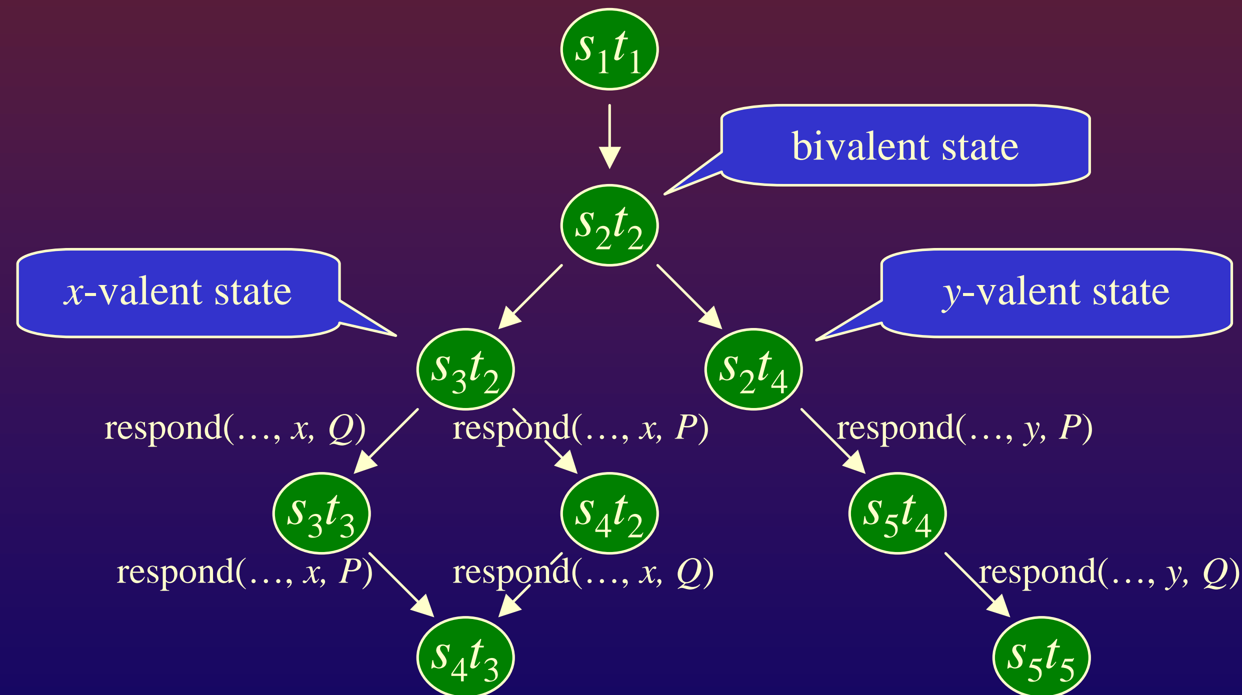
- A *consensus protocol* is a set of processes that each take an input value and then decide on one of the values as a common value that they all return



- X solves n-process consensus if a consensus protocol $\{F_1,\ldots,F_n$;[auxiliary *R/W regs*], $X\}$ exists
- The largest such n is the *consensus number* for X

Bivalent and *x*-valent States

- Given a consensus protocol, we say that it is in a
  - *bivalent state* if it could still decide *x* or *y*
  - *x-valent state* if it can only end up deciding *x*



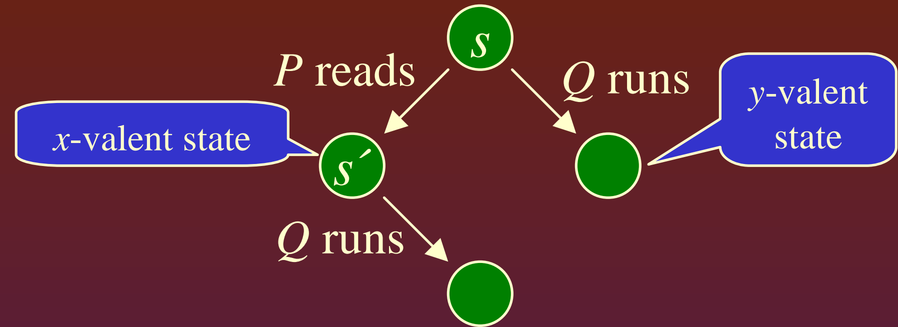
Atomic R/W Registers

Atomic R/W registers have consensus number 1

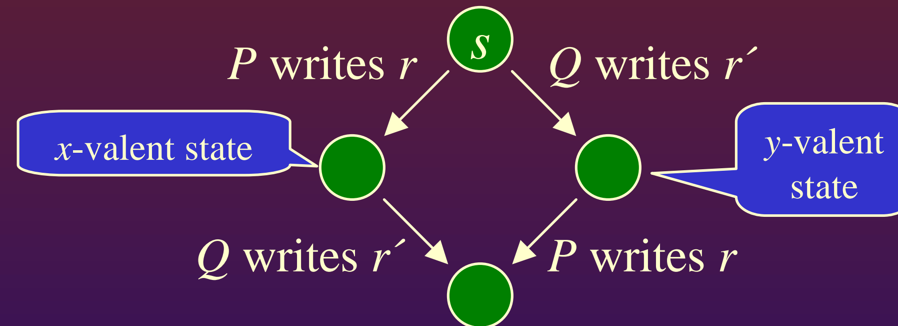
- Proof:
  1. Suppose a protocol for 2 processes *P* and *Q* exists
  2. Let *P* run until next operation would decide *x*
  3. Let *Q* run until next operation would decide *y*
  4. Repeat 2-3 until *P* would decide *x* and *Q* decide *y*
  5. Consider the three cases:
     1. *P* reads a shared atomic R/W register
     2. *P* and *Q* write to different atomic R/W registers
     3. *P* and *Q* write to the same atomic R/W register

Atomic R/W Registers

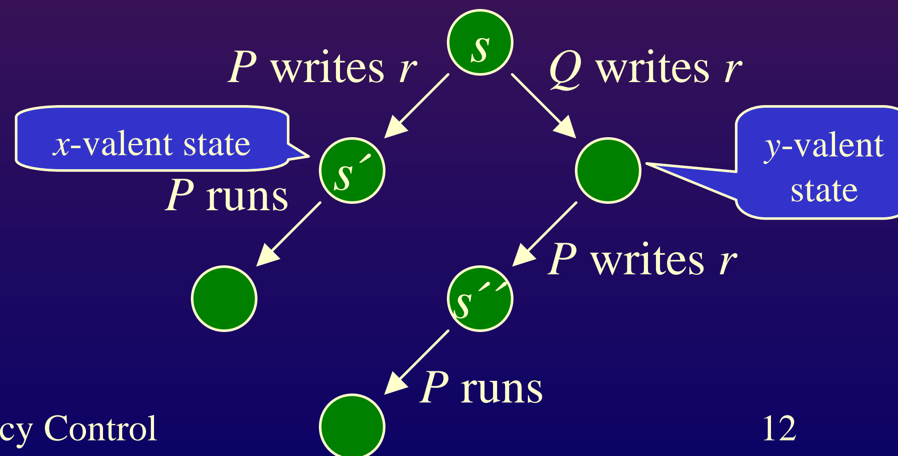
1. *P* reads a register leads to a contradiction: *Q* cannot know that *P* has decided *x*



2. *P* writes *r* (and decides *x*) and *Q* writes *r´* leads to a contradiction: Opposite order would lead to the same state but decide *y*



3. *P* and *Q* write *r* leads to a contradiction: *P* overwrites *r* and cannot know whether *Q* has written *r* and decided *y*



Read-Modify-Write Operations

RMW-registers have consensus number *n* ? 2

- Proof: Running this decision protocol with register *r* initialised to *v*, where *f* (*v*) ? *v*, achieves 2-process consensus

RMW (*r* : register, *f* : function) {
  previous := *r*
  r ::= *f* (*r*)
  return *previous*
}

P.decide (*input*) {
  prefer[*P*] := *input*
  if RMW (*r*, *f*) = *v* **then**
    return *prefer*[*P*]
  else
    return *prefer*[*Q*]
}

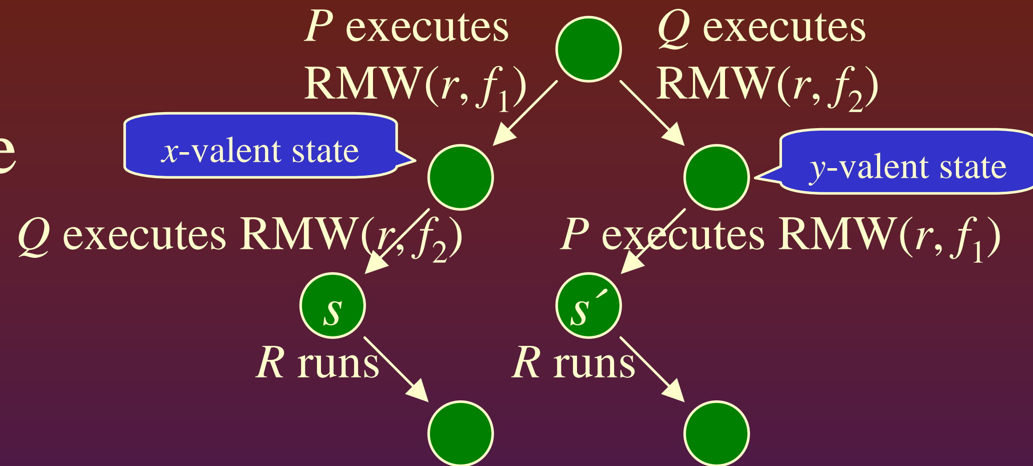
Read-Modify-Write Operations

read-, write-, test&set- swap- and fetch&add-registers have consensus number $n < 3$

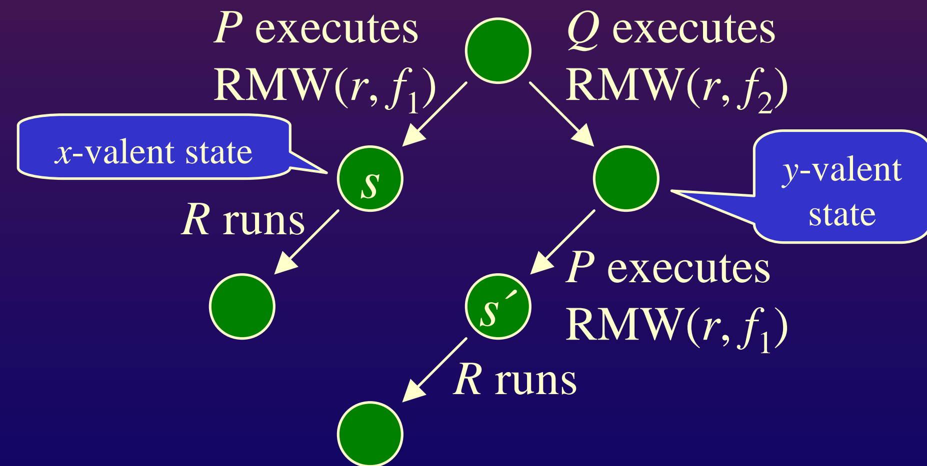
- For all pairs (f_1, f_2) of these functions, either
  - $f_1(f_2(v)) = f_2(f_1(v))$ (they commute), or
  - $f_1(f_2(v)) = f_1(v)$ (f_1 overwrites), or
  - $f_2(f_1(v)) = f_2(v)$ (f_2 overwrites)
- Proof: Assume three processes *P*, *Q*, *R*. Run *P* and *Q* until next operation would achieve consensus between *P* and *Q* as before. Consider the two cases:
  1. f_1 and f_2 commute
  2. f_1 overwrites

Read-Modify-Write Operations

1. If f_1 and f_2 commute, R cannot tell from the state which of P and Q executed first



2. If f_1 overwrites, R cannot tell from the state whether Q has executed before P



Read-Modify-Write Operations

compare&swap-registers have infinite consensus number

```
cmp&swp (r : register,
            old : value , new : value) {
  previous := r
  if previous = old then r ::= new
  return previous
}
```

- Proof: Running this decision protocol achieves n-process consensus for unbounded n

```
P.decide (input) {
  first := cmp&swp (r, ? , input)
  if first = ?  then
    return input
  else
    return first
}
```

FIFO Queues

FIFO queues have consensus number n ? 2

- Proof: Running this decision protocol with a queue q initialised by enqueueing first 0 and then 1 achieves 2-process consensus

```
P.decide (input) {
   prefer[P] := input
   if deq(q) = 0 then
      return prefer[P]
   else
      return prefer[Q]
}
```

FIFO Queues

FIFO queues have consensus number $n < 3$

- Proof: Assume three processes P, Q, R. Run P and Q until next operation would achieve consensus between P and Q as before. Consider the three cases:
  1. Both P and Q execute deq(q)
  2. P executes deq(q) and Q enq(q)
  3. Both P and Q execute enq(q)

FIFO Queues

1. If both execute deq(q), R cannot tell which of P and Q dequeued first
2. If P executes enq(q) and Q deq(q) then
   - If q was nonempty, R cannot tell which occurred first
   - If q was empty, R cannot tell whether Q executed deq(q) before P executed enq(q)
3. If both execute enq(q), run P until it dequeues, then Q until it dequeues. Now R cannot tell whether P or Q enqueued first

Augmented Queues

- An *augmented queue* is a queue augmented with a *peek*(*q*) operation which returns the first element

Augmented queues have infinite consensus number

- Proof: Running this protocol achieves *n*-process consensus for unbounded *n*

```
P.decide (input) {
   enq(q,input)
   return peek(q)
}
```

Universal Objects

- A *universal object* is an object that can implement any other object (wait-free for n processes)

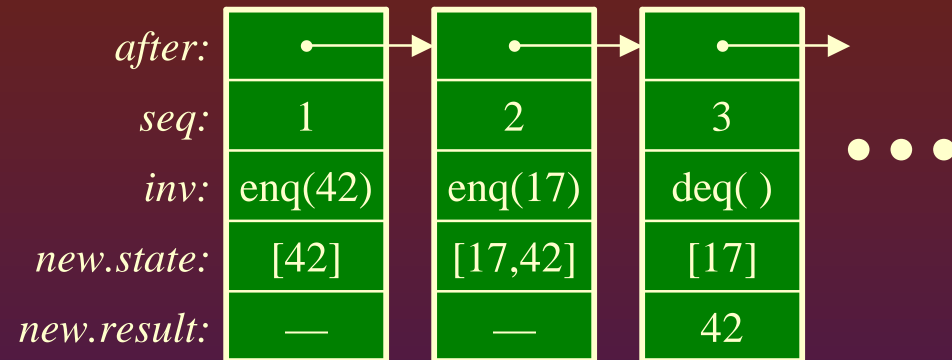
Any object with consensus number n is a universal object for n processes

- Proof: We give an implementation based only on
  - Atomic R/W registers, and
  - A consensus protocol for n processes.

Representing an Object

- We represent an object by a linked list of cells of invokations, resulting states and values
- State transitions are given by a relation apply : (inv×state)? (state×result)
- The array *head*[*P*] holds the last cell seen by process *P*

Execution history of a queue:



apply (enq(42), []) = ([42], —)
apply (enq(17), [42]) = ([17,42], —)
apply (deq( ), [17,42]) = ([17], 42)

P	*Q*	*R*
2	2	3

Simulating an Object

1. When process *P* calls universal(*what*), a new cell *mine* is created and announced
2. A recently added cell *c* (e.g., the last) is selected
3. If another process' announced cell, *help*, is not in the list yet, this cell is preferred over *mine*
4. The preferred cell is linked onto *c* if *c* really *is* last
5. The next state is computed if it has not yet been computed
6. When *mine* has been added to the list, *mine.new.result* is returned

```
  universal (invokation what) {
1     mine = { seq = 0,
               inv = what,
               new = ConsensusObject.create ( ),
               after = NULL }
      announce [P] := mine
2     for each process Q do
         head [P] := max (head [P], head [Q])
      while announce [P].seq = 0 do {
         cell* c := head [P]
3        cell* help := announce [(c.seq mod n) + 1]
         if help.seq = 0 then
            prefer := help
         else
            prefer := announce [P]
4        d := decide (c.after, prefer)
5        decide (d.new, apply (d.inv, c.new.state))
         d.seq := c.seq + 1
         head [P] := d
      }
      head [P] := announce [P]
6     return announce [P].new.result
  }
```

Simulation Properties

- Every *mine* cell is eventually added to the list, either by *P* or some other process
- By helping another process first, the simulation is *bounded wait-free*: *P* can execute the **while**-loop at most $n + 1$ times before the *mine* cell has been added

```
  universal (invokation what) {
1   mine = { seq = 0,
             inv = what,
             new = ConsensusObject.create ( ),
             after = NULL }
    announce [P] := mine
2   for each process Q do
      head [P] := max (head [P], head [Q])
    while announce [P].seq = 0 do {
      cell* c := head [P]
3     cell* help := announce [(c.seq mod n) + 1]
      if help.seq = 0 then
        prefer := help
      else
        prefer := announce [P]
4     d := decide (c.after, prefer)
5     decide (d.new, apply (d.inv, c.new.state))
      d.seq := c.seq + 1
      head [P] := d
    }
    head [P] := announce [P]
6   return announce [P].new.result
  }
```

Wait-Free Implementation Hierarchy

Consensus number	*Object*
1	atomic R/W registers
2	test&set, swap, fetch&add, queue, stack
:	:
$2n$-2	n-register assignment
:	:
?	memory-to-memory move and swap, augmented queue, compare&swap, fetch&cons, sticky byte

- An object at level n can implement an object at level m ? n
- No object at level n can implement an object at level $m > n$