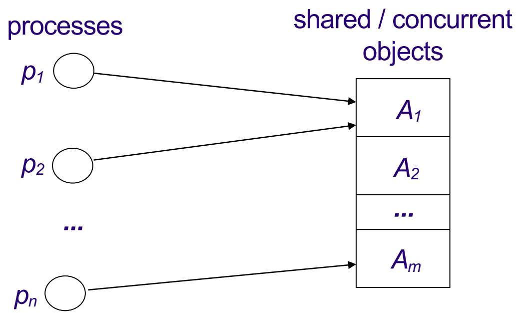
11. Shared Memory Objects

Introduction

- Concurrent systems or shared memory systems
  - Composed by processes + shared objects (aka concurrent objects)
  - Processes communicate exclusively through the shared objects
  - It is a system model alternative to *message-passing*



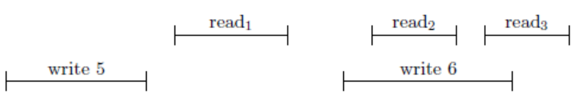
Concurrent objects

- Examples of shared objects:
  - Registers – 2 operations: read, write
  - Test&set – 1 operation: test&set
  - Compare&swap – 1 operation: compare&swap
  - FIFO queue – 2 operations: enq, deq
  - Arrays of the previous, …

- What are they in real-life?
  - Programming language constructs for multi-threading
  - Parallel computers' shared memory and operations
  - Emulations in distributed systems

Registers

- Classification via number of readers/writers:
  - single- or multi-reader
  - single- or multi-writer
- Semantics with concurrent operations:
  - **Safe**: a read not concurrent with any write obtains the correct value (otherwise any outcome is possible)
  - **Regular**: safe + read that overlaps a write obtains either the old or new value
  - **Atomic**: safe + reads and writes behave as if they occur in some definite order



$read_2$ / $read_3$ can return 6 / 5 in a regular register (and, e.g., 42 in a safe register), but not in an atomic register

Two important questions

- How can shared memory objects be emulated on distributed systems?
  - i.e., on top of message-passing system models, e.g., the object(s) can be implemented by a (set of) server(s) accessed through RPC (e.g., using quorums for storage or consensus algorithms for implementing SMR)
- What is the "synchronization power" of each shared memory object?
  - *Maurice Herlihy, "Wait-free Synchronization", ACM TPLS, 1991*

    (This lecture!)

Wait-free concurrent objects

- Wait-free implementation of a concurrent data object
  - Guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds on the other processes
- The wait-free property provides fault tolerance
  - No process can be prevented from completing an operation by undetected halting failures of other processes, or by arbitrary variations in their speed
- Fundamental problem of wait-free synchronization:
  - Given two concurrent objects X and Y, is there a wait-free implementation of X by Y?
  - E.g., Is it possible to implement a wait-free atomic register with a safe register? Is it possible to implement a FIFO queue using atomic registers?

    (Yes and No, respectively)

Synchronization power of shared objects

- Given two concurrent objects X and Y, does a wait-free implementation of X by Y exist?
  - If not, **X is more "powerful" than Y**
- Herlihy defines synchronization power in terms of **consensus number**
  - Each object has an associated consensus number
  - **The consensus number for X is the largest *n* for which X solves consensus among *n* processes**
    - In an asynchronous system in which up to *n-1* processes may crash
  - If no largest ***n*** exists, the consensus number is said to be infinite

Consensus number

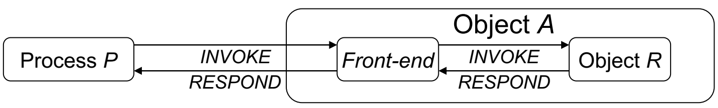
consensus number	object
1	**register**
2	***test&set***, *swap*, *fetch&add*, *queue*, *stack*
…	…
2n – 2	atomic n-register assignment
…	…
∞	***compare&swap***, *m2m move*, *m2m swap*, *augmented queue*, *fetch&cons*, *sticky byte*

Model

- **Concurrent system**: *[P_1,..., P_n; A_1,..., A_m]*
  - P_i : processes that invoke operations on the shared objects
  - A_i: shared objects
- Environment:
  - Asynchronous system
  - Up to *n-1* processes can **crash** but objects are correct
- Additional Assumption: **Linearizability** of all objects
  - Although operations of concurrent processes may overlap, each operation appears to take effect instantaneously at some point between its invocation and response
  - In particular, operations that do not overlap take effect in their "real-time" order
  - The history "appears" sequential to each individual process
  - This apparent sequential interleaving respects the real time precedence of operations (similar to an atomic register)

Model

- An implementation of an object A is a concurrent system *{F1,...,Fn; R}*, where the F's are the **front-ends** and R is called the **representation object**.
  - *R* is the object that implements *A*
  - F_i is the procedure called by process P_i, to execute an operation
- We say that R implements A if there exists a wait-free implementation *{F1,...,Fn; R}* of A



Consensus protocol

- Consensus protocol
  - System of *n* processes that communicate through a set of shared objects *{X1, ... Xm}*
  - Each process starts with an input value from some domain
  - They communicate by doing operations to the shared objects
  - They eventually agree on a common input value and halt

- A consensus protocol is required to be:
  - **Consistent**: distinct processes never decide on distinct values
  - **Wait-free**: each process decides after a finite number of steps
  - **Valid**: the common decision value is the input to some process

  (these are different names for properties we know)

Consensus object

- A wait-free linearizable implementation of a consensus object is called a consensus protocol

- Consensus object (using abstract data types term.):
  - decide(input: value) returns(value)

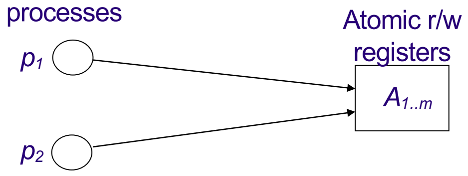
Impossibility: consensus number

- Recall the notion of **consensus number**:
  - The consensus number for X is the largest ***n*** for which X solves ***n***-process consensus
  - If no largest ***n*** exists, the consensus number is said to be infinite.
- Theorem
  - If X has consensus number n,
  - and Y has consensus number $m < n$,
  - then there exists no wait-free implementation of X by Y
  - in a system of more than m processes.

Impossibility: registers

- Theorem: Read/write registers have consensus number 1.
  - What does it mean to have consensus number 1?
  - Why is the theorem true?

    Think about the simplest case:



Impossibility: registers

- Corollary:
  - It is impossible to construct a wait-free implementation of any object with consensus number greater than 1 using atomic read/write registers.
- Registers have no consensus power so they appear freely in the constructions that follow

Impossibility: Read-Modify-Write (RMW)

- Theorem:
  - A register with any nontrivial read-modify-write operation has a consensus number <u>at least</u> 2
- Examples of nontrivial rmw
  - <u>test&set</u>, swap, fetch&add
  - Nontrivial means the function performed is not identity

2-consensus with test&set

- test&set implementation (atomic)

```
test&set(v:value) returns(value)
   previous := v
   v := 1        //modifies local value
   return previous
 end test&set
```

- 2-consensus implementation with test&set

```
shared prefer[1..2] := [⊥,⊥]; v := 0

decide(input:value) returns(value)
   prefer[p] := input
   if test&set(v) = 0
     then return prefer[p]
     else return prefer[q]
   end if
 end decide
```

This is the code for process p; for process q change every p for q and vice-versa

Theorem: test&set has consensus number 2

Impossibility: test&set

- Theorem: There is no wait-free solution to three-process consensus using any combination of test&set operations
  - Generalization: No solution for any combination of read-modify-write operations that apply functions that either
    - Commute - f1(f2(v)) = f2(f1(v)), or
    - One function overwrites the other - f1(f2(v)) = f1(v)
  - This includes test&set, swap and fetch&add

Compare&swap (CAS)

- CAS implementation (atomic)

```
CAS(v:value, old:value, new:value) returns(value)
  previous := v
  if previous = old then v := new endif
  return previous
end CAS
```

- Wait-free *n*-consenus implementation using CAS

```
shared   v := ⊥
decide(input:value) returns(value)
  first := CAS(v, ⊥,input)
  if first = ⊥ then return input
  else return first endif
end decide
```

- **Theorem**: A compare&swap register has infinite consensus number

Some lessons learned

- Consensus numbers:
  - Registers: 1
  - Test&set: 2
  - Compare&swap: *n*
- Using … we can implement …:
  - Compare&swap: Test&set
  - Test&set: Registers
  - Compare&swap: Registers
- Using … we cannot implement…:
  - Test&set: compare&swap
  - Registers: Test&set
  - Registers: Compare&swap

Queues…

- **FIFO Queue**
  - enq(queue,value)
  - deq(queue):value
- Algorithm

```
shared prefer[1..2] := [⊥,⊥];
       queue := (0)

decide(input:value) returns(value)
       prefer[p] := input
       if deq(queue) = 0
         then return prefer[p]
         else return prefer[q]
       end if
     end decide
```

- **Theorem:** FIFO Queue has consensus number 2

- **Augmented Queue**
  - enq(queue,value)
  - deq(queue):value
  - peek(queue):value
- Algorithm

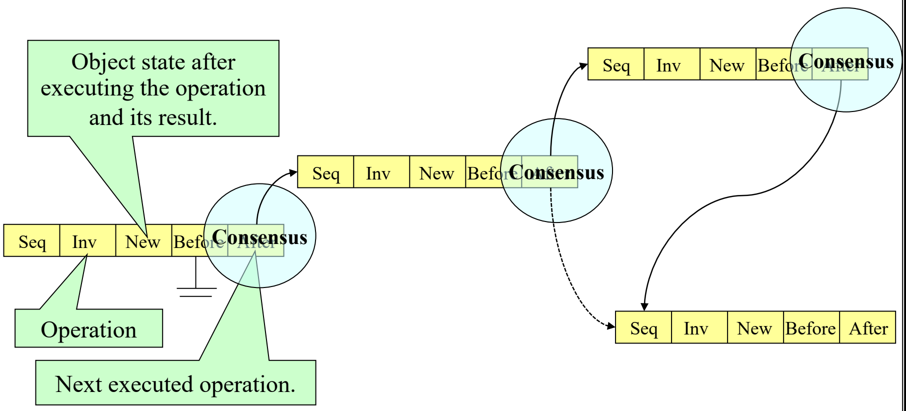
```
shared queue := ()

decide(input:value) returns(value)
       enq(queue,input)
       return peek(queue)
end decide
```

- **Theorem:** Augmented Queue has infinite consensus number

Universality results

- An object is **universal** if it implements any other object
- Any object with consensus number **n** is universal in a system of **n** (or fewer) processes
  - E.g., compare&swap is universal
- Basic idea for a universal construction:
  - Represent the object as a linked list, where the sequence of cells represents the sequence of operations applied to the object
  - A process executes an operation by threading a new cell on to the end of the list
  - When the cell becomes sufficiently old, it is reclaimed and reused.
  - In other words: consensus is used to define a total order on the operations, i.e., to establish the pointer to the next element on the list

Universal Construction



Remarks

- "Wait-free Synchronization" is considered one of the most influent papers in distributed computing
  - 2003 Dijkstra Prize (distributed computing)
  - 2004 Gödel prize (theoretical computer science)
- Sort of Periodic Table (Chemistry) for shared memory distributed systems
- High impact: e.g., parallel computers now implement *compare&swap*, not *test&set*