
# Lecture 19: Wait-free Hierarchy

CS 539 / ECE 526

Distributed Algorithms

# Atomic Objects

- We have seen many (atomic) objects:
  - Many types of shared (read/write) registers
  - Test-and-Set
  - Read-Modify-Write
  - Atomic queue
  - ……
- Can we use one type of object to implement another?

# Generic Method 1

- Mutual exclusion gives a way to implement *any* atomic object
  - Mutex can be solved with atomic or safe read/write registers
- Ensures all operations are *sequential*, hence non-overlapping and linearizable
- Downside: slow and not fault tolerant

# Today

- Can we use one type of object to implement another with wait-freedom?
  - Wait-free == tolerate all but one crashes
  - We have seen wait-free implementations of "stronger" registers using "weaker" ones
  - Are there wait-free implementations of other (atomic) objects (e.g., atomic queues, RMW)?

# Outline

- Wait-free hierarchy
- Consensus numbers of
  - Read/write registers
  - Queues
  - Compare-and-Swap

# Generic Method 2

- Use a consensus / replication algorithm to agree on the sequence of operations
  - Replicate the object at every process
  - Agree (totally order) all the operations
  - Apply operations locally in the agreed order

- Wait-free if consensus algo is wait-free
  - But is wait-free consensus itself possible?

# Wait-free Hierarchy [Herlihy, 1991]

- If object Y can be used to solve wait-free consensus in some setting but object X cannot, then there exists no wait-free implementation of Y from X in that setting
  - Note: consensus means agreement in the discussion of wait-free hierarchy

# What Settings?

- We already specified most of the model:
  - Shared memory
  - Asynchrony
  - Wait-free (tolerate all but one crash)
  - Let's also restrict to deterministic case
- Number of processes turn out to be key!

# Wait-free Hierarchy [Herlihy, 1991]

- If object Y can be used to solve wait-free consensus with *n* processes but object X cannot, then there exists no wait-free implementation of Y from X with *n* procs

# Consensus Number [Herlihy, 1991]

- Definition: The consensus number of an object is the largest number *n* for which *n*-proc wait-free consensus can be solved using that object plus registers
  - *n*-proc solvable, (n+1)-proc unsolvable
  - Can use many instances of X plus multi-valued MRMW read/write atomic registers

# Consensus Number of Common Objects

| Objects | Consensus Number |
| --- | --- |
| Read/write atomic registers | 1 |
| Queue, stack, Test-and-Set, … | 2 |
| Compare-and-Swap, queue with peek, … | ∞ |

# Wait-free Hierarchy

- If object Y has consensus number *n* and object X has consensus number $m < n$, there exists no wait-free implementation of Y from X and registers in a system with $>m$ processes

# Outline

- Wait-free hierarchy
- Consensus numbers of
  - Read/write registers
  - Queues
  - Compare-and-Swap

# CN of Read/Write Registers = 1

- CN of any object is at least 1 because one-proc consensus is trivial

- Theorem 1: there exists no wait-free consensus algorithm for two processes using only read/write atomic registers.

# Recall Configurations

- Union of the states of all parties
- A protocol execution is an evolution of configurations: $C_0 \rightarrow C_1 \rightarrow C_2 \ldots$
- In async msg passing, config evolves after each msg arrival

# Configurations in Shared Memory

- Union of states of **all procs & all objects**
- In async shared memory with atomic objects, config evolves after every atomic operation
- As before, for *concurrent* operations, the order to apply them does not matter
- One operation *happens before* another if
  - They are performed by the same process; or
  - They access the same object, and at least one of them *updates* the object; or
  - Due to transitivity

# Recall Valency

- A config C is **0-valent**, if in all configs reachable from C, processes decide 0
  - No matter what happens from now on, decide 0
- A config C is **1-valent,** if ……, all decide 1
- **Univalent** = 0-valent or 1-valent
- **Bivalent** = not univalent

# Critical Configuration

- A configuration is *critical* if it is bivalent, and all its successor configurations are univalent
  - In general, may not exist, hence uninteresting
  - But must exist in a wait-free consensus algorithm!

- Alternative definition of wait-free: every process terminates in finite number of steps

# Critical Configuration

- Lemma: Every wait-free consensus algorithm has a critical configuration.
  - Proof: Suppose not. Every bivalent config has a bivalent successor config
  - There exists an initial bivalent config (Lecture 8)
  - There is an infinite run
  - At least one process took infinitely many steps
  - Not wait free. QED.

# CN of Read/Write Registers = 1

- Theorem 1: no wait-free consensus algorithm for 2 procs using only r/w atomic registers.
  - Proof: Suppose for contradiction that there is such an algorithm for two processes A and B
  - Let it run to a critical config S
  - The next op by A or B leads to univalency
  - Critical config S is bivalent → has both evolutions
  - WLOG, $S \rightarrow_A$ 0-valent, $S \rightarrow_B$ 1-valent
  - Let us consider these two next ops of A and B

# CN of Read/Write Registers = 1

- Critical config S, S $\rightarrow_A$ 0-valent, S $\rightarrow_B$ 1-valent
- Cannot be concurrent ops
- Must access same register & at least one is write

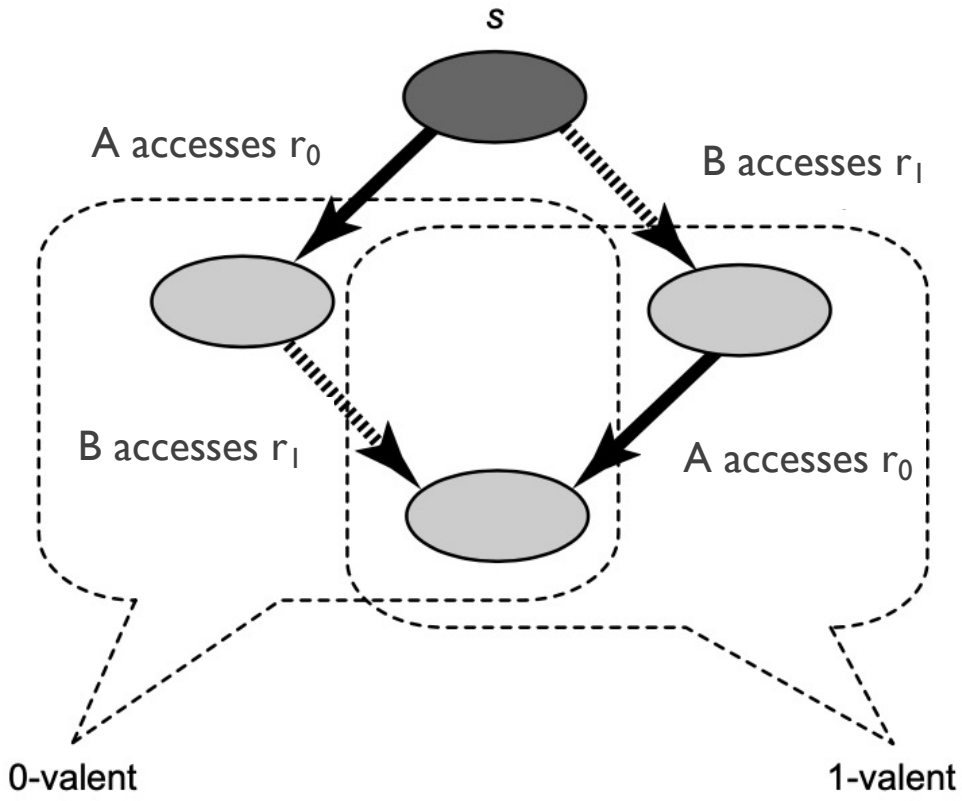



Figure from Herlihy and Shavit: *The Art of Multiprocessor Programming*

# CN of Read/Write Registers = 1

- Critical config S, S $\rightarrow_A$ 0-valent, S $\rightarrow_B$ 1-valent
- Case 1: one process reads, (WLOG A reads)

A's read has no effect on B

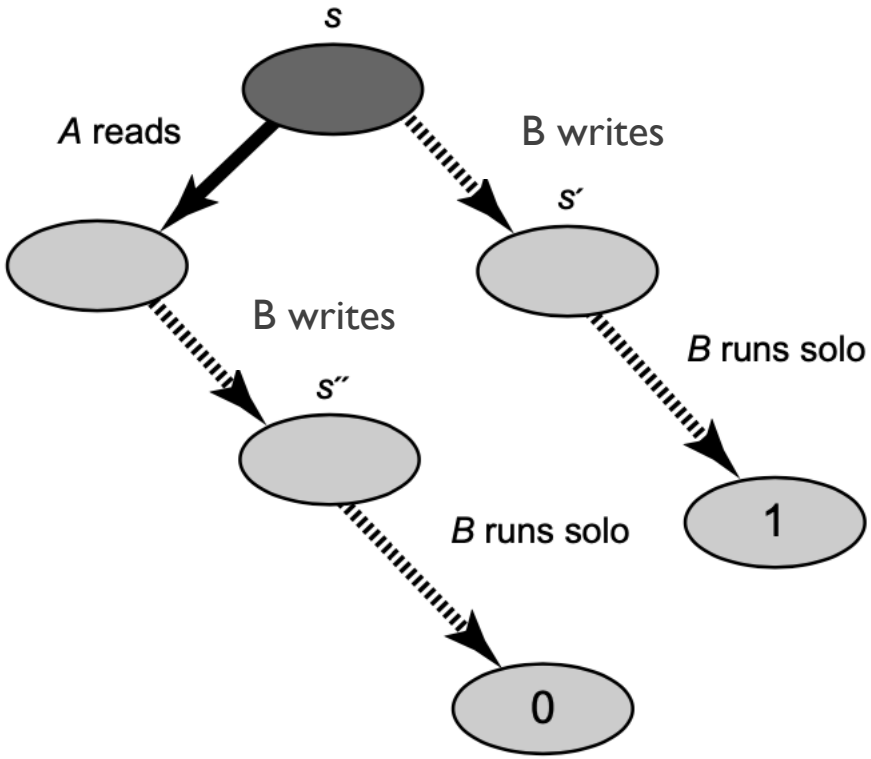
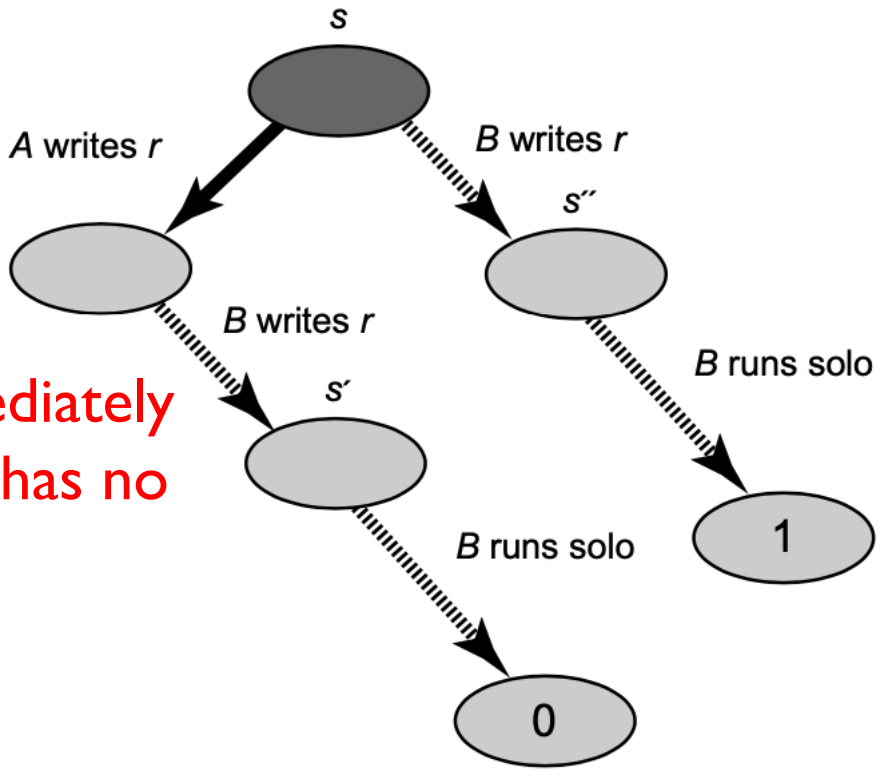



Figure from Herlihy and Shavit: *The Art of Multiprocessor Programming*

# CN of Read/Write Registers = 1

- Critical config S, S $\rightarrow_A$ 0-valent, S $\rightarrow_B$ 1-valent
- Case 1: one process reads, (WLOG A reads)
- Case 2: both write same register




A's write is immediately overwritten and has no effect

Figure from Herlihy and Shavit: *The Art of Multiprocessor Programming*

# CN of Read/Write Registers = 1

- Theorem 1: no wait-free consensus algorithm for 2 procs using only r/w atomic registers.
  - Proof: Suppose for contradiction that there is such an algorithm for two processes A and B
  - Let it run to a critical config S
  - The next op by A or B leads to univalency
  - Critical config S is bivalent → has both evolutions
  - WLOG, $S \rightarrow_A$ 0-valent, $S \rightarrow_B$ 1-valent
  - We considered these two next ops of A and B and got contradictions in all cases, QED

# Outline

- Wait-free hierarchy
- Consensus numbers of
  - Read/write registers
  - Queues
  - Compare-and-Swap

# Atomic Queue

- Same interface of basic queue
  - Supports atomic enqueue() and dequeue() by at least two processes
    - Does not support peek()
  - dequeue() on an empty queue returns $\perp$

# CN of Queue = 2

- Theorem 2.1: there is a wait-free consensus algorithm for two processes using a queue (assuming required initial state)

- Theorem 2.2: there is no wait-free consensus algorithm for three processes using queues and atomic read/write registers

# CN of Queue ≥ 2

- Theorem 2.1: wait-free 2-proc consensus algo using a queue (with good initial state)

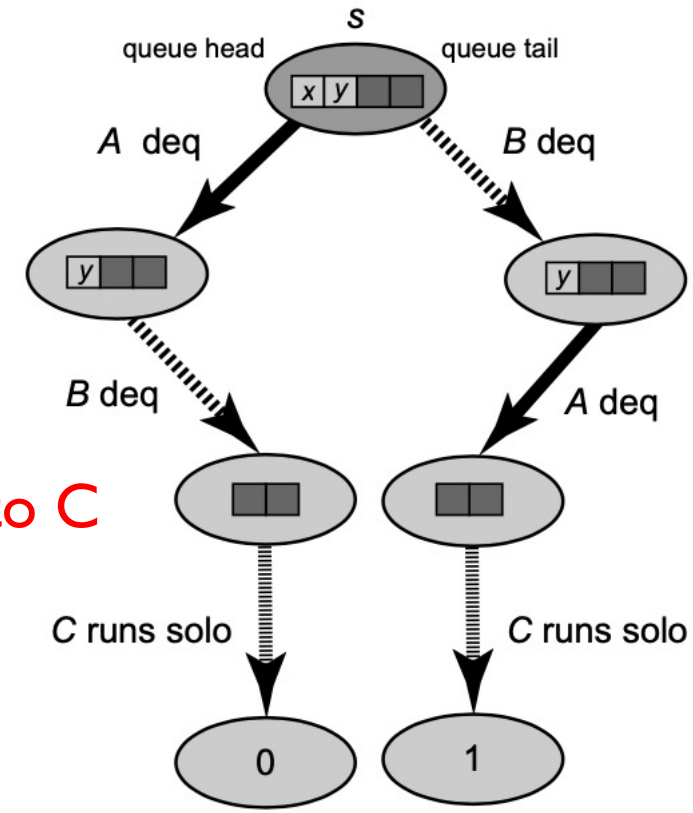
```
// Queue initialized with one element, two SW register
Initially Q = ["winner"],  Prefer[2] = [⊥, ⊥];

// code for process i, with input x_i
Prefer[i] = x_i
if dequeue(Q) == "winner"
        output Prefer[i];
else output Prefer[1-i];
```

# CN of Queue ≤ 2

- Theorem 2.2: no wait-free 3-proc consensus algo using queues and r/w atomic registers
  - Proof: Suppose for contradiction that there is such an algorithm for three processes A, B, and C
  - Critical config S, WLOG, S $\rightarrow_A$ 0-valent, S $\rightarrow_B$ 1-valent
  - These 2 next ops of A and B cannot be concurrent
  - If accessing same register, same proof as before
  - Must access the same queue, and at least one of them updates the queue
    - Both enqueue and dequeue update the queue

# CN of Queue ≤ 2

- Critical config S, S $\rightarrow_A$ 0-valent, S $\rightarrow_B$ 1-valent
- Case 1: A and B both dequeue




Indistinguishable to C

Figure from Herlihy and Shavit: *The Art of Multiprocessor Programming*

# CN of Queue ≤ 2

- Critical config S, S $\rightarrow_A$ 0-valent, S $\rightarrow_B$ 1-valent
- Case 1: A and B both dequeue
- Case 2: A enqueues and B dequeues (or vice versa)
  - If queue is not empty, end results are the same
  - If queue is empty,

    A enqueue == B dequeue then A enqueue
  - In either case, C cannot distinguish if it runs solo

# CN of Queue ≤ 2

- Critical config S, S $\rightarrow_A$ 0-valent, S $\rightarrow_B$ 1-valent
- Case 1: A and B both dequeue
- Case 2: A enqueues and B dequeues (or vice versa)
- Case 3: A and B both enqueue
  - C *can* distinguish the two resulting configs
  - … if and only if C dequeues
  - Same applies to A and B!

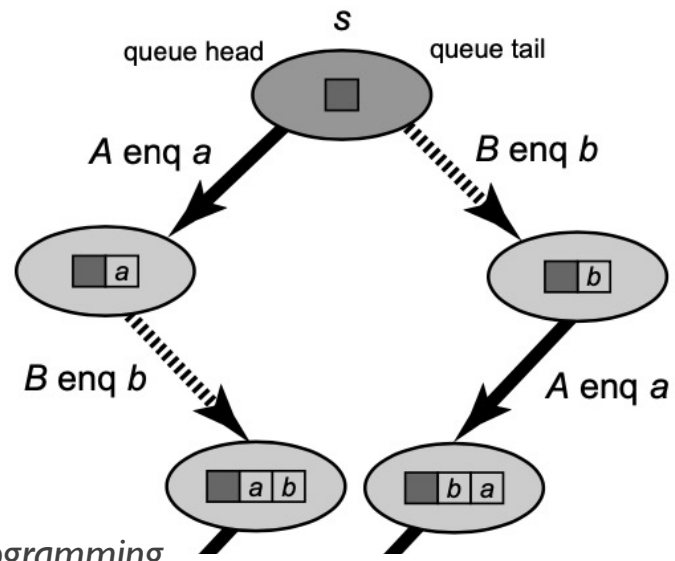



Figure from Herlihy and Shavit: *The Art of Multiprocessor Programming*

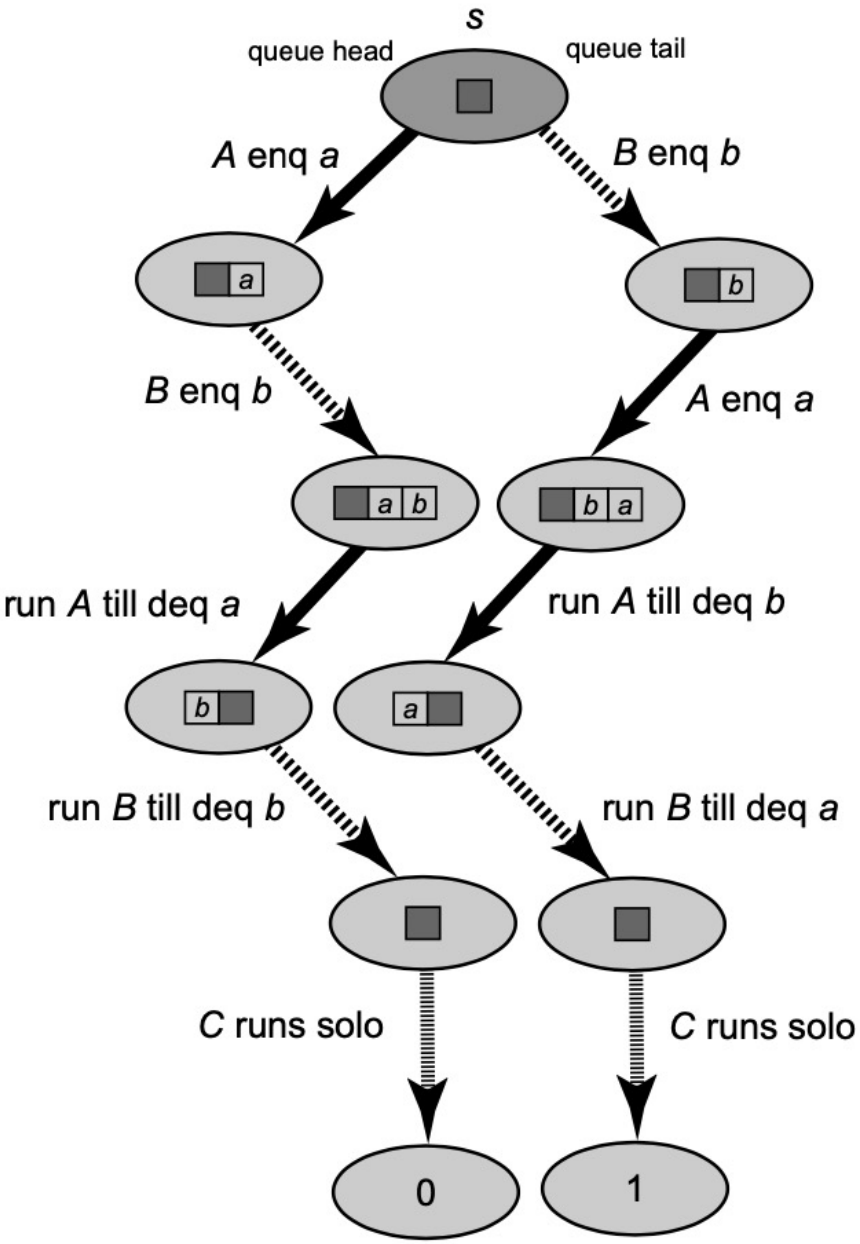



Figure from Herlihy and Shavit: *The Art of Multiprocessor Programming*

# CN of Queue ≤ 2

- Theorem 2.2: no wait-free 3-proc consensus algo using queues and r/w atomic registers
  - Proof: Suppose for contradiction that there is such an algorithm for three processes A, B, and C
  - Critical config S, WLOG, S $\rightarrow_A$ 0-valent, S $\rightarrow_B$ 1-valent
  - These 2 next ops of A and B cannot be concurrent
  - If accessing same register, same proof as before
  - Must access the same queue, and at least one of them updates the queue
    - Both enqueue and dequeue update the queue
  - Got contradictions in all cases, QED

# Outline

- Wait-free hierarchy
- Consensus numbers of
  - Read/write registers
  - Queues
  - Compare-and-Swap

# Compare-and-Swap

- Interface
  - Stores a value V, supports read(V), and the following atomic operation

compare&swap(V, old, new):

```
tmp = V
if (tmp == old)
        V = new
return tmp
```

# CN of Compare-and-Swap = ∞

- Theorem 3: there is a wait-free consensus algo for *n* procs for all *n* using Compare-and-Swap

```
Initially V = ⊥        // Compare-and-Swap object

// code for process i, with input x_i ≠ ⊥
val = compare&swap(V, old=⊥, new=x_i)
if (val == ⊥)          // process i is 1st
      output x_i
else
      output val
```

# Summary

- Wait-free hierarchy answers if object Y has wait-free implementation from object X
- Classify "strength" using consensus numbers: max # of procs for solving wait-free consensus

| Objects | Consensus number |
|---|---|
| Read/write atomic registers | 1 |
| Queue, stack, Test-and-Set, … | 2 |
| Compare-and-Swap, queue with peek, … | ∞ |