The consensus hierarchy

Reference: [Her91] Maurice Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages and Systems, 13(1), pages 124-149, January 1991.

Solving consensus in different kinds of shared memory systems.

Model: Asynchronous system, any number of halting failures.
Using read/Write registers $\longrightarrow$ impossible.
Using Compare and Swap registers $\longrightarrow$ solvable.

What if system has register and FIFO queues?

Assume we can initialize shared objects to any state we like. Initialize the queue with 1 element "winner" and 1 element "loser" (with "winner" ahead of "loser" in the queue).

Algorithm for 2 processes

Write your input in your register.

dequeue from queue.

If dequeue returns winner, return your own input value.

else /* dequeue returned loser */

return the other process's input value (obtained from its register).

Every process returns the input value of the first process to dequeue, thus ensuring the validity condition.

What if n = 3 ?

THEOREM.

There is no algorithm for 3-process consensus using FIFO queues and Read/Write Registers.

PROOF.

By contradiction. Let's assume there exists an algorithm that can solve consensus for three processes, P,Q,R. Recall the proof of last day, we know there exists a multi-valent configuration C (called critical config) such that any process's single step will take the system into a 0-valent or 1-valent configuration.

Then we show there is no such C in this model. At first, we know in critical state, all the processes must access the same queue object, as in the proof of the last lecture. Suppose a step by P makes the system 0-valent, while a step by process Q would make the system 1-valent. The following part is a case analysis.

Case I: Both P and Q execute *dequeue* operations.

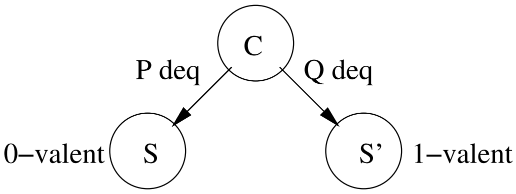

Figure 1: Case I: Both P and Q execute *dequeue* operations.

Let s be the configuration after P executes *deq* and let s' be the configuration after Q does the same thing. If process R runs by itself it should eventually decide on 0 if it starts from s and 1 if it starts from s'. But the content of *queue* are identical in both states, so the configuration of s and s' differ only in the internal states of P and Q. Also because both P and Q can become inactive immediately after they executes *deq* operation, they cannot tell any info to process R. Then state s and s' are same to process R, and it should decide on the same value in both s and s', which contradicts that s is 0-valent and s' is 1-valent.

Case II: P executes *dequeue* and Q executes *enqueue*.

Case IIa: The queue is non-empty in C.

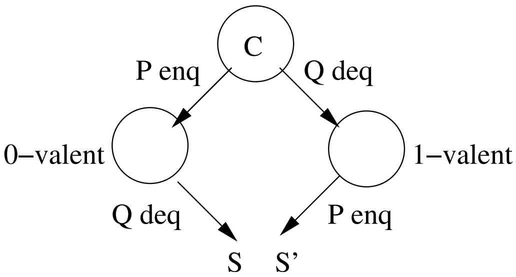

Figure 2: Case IIa: One executes *dequeue* and the other executes *enqueue* and the queue is non-empty in C.

If the queue is non-empty in C, then let S be the configuration if P *enqueues* and then Q *dequeues*, and let S' be the configuration if Q *dequeue* and then P *enqueues*. Because both process P and Q get the same value or response in the two executions, and the state of the queue is also identical, $s = s'$, which contradicts that s is 0-valent and s' is 1-valent.

Case IIb: The queue is empty in C.

If the queue is empty in C, then let S be the configuration if P *enq* then Q *deq*, and let S' be the configuration if only Q *deq*, because in each case the queue is empty, they are indistinguishable to R since S and S' differ only in the internal state of P and Q, both of which can be killed immediately after executing *deq* or *enq* operation.

Case III. P dequeues and Q enqueues

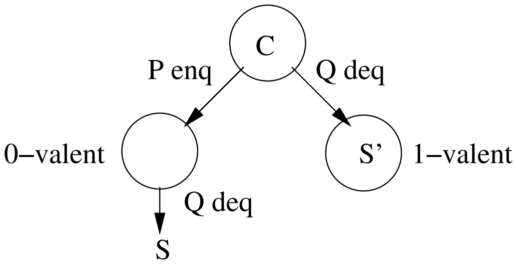

Figure 3: Case IIb: P executes *dequeue* and Q executes *enqueue* and the queue is empty in C.

Symmetric to Case II.

Case IV. Both P and Q execute *enqueue* operations.

Let s be the state after the following operations:

(1) P and Q enqueue items v and w in that order.
(2) Let P run *deq* until it dequeues v. (P can't halt before doing this; see below.)
(3) Run Q until it dequeues w. (Again, Q cannot halt before doing this.)

Let s' be the state after the following operations:

(1) Q and P enqueue items w and v in that order.
(2) Run P until it dequeues w.
(3) Run Q until it dequeues v.

Since the only way to obesrve the order of the two *enq*s done by P and Q is through *deq*ing until one of them items is removed, P and Q cannot halt before doing so. The operations and states of P and all the other processes are identical until process P dequeues v or w. Since P can be killed before it can tell anyone else, $Q's$ states and operations are also identical until it dequeues v or w. After processes Q was killed immediately after he knows the truth, all the evidence of the order of *enq* was eliminated.

Now, R can be run from s or s', and it will not be able to tell the difference between them. So it must output the same result starting from both configurations, a contradiction.

Definition.

The consensus number for object type T (denoted cons(T)) is n if there exist a wait-free algorithm for consensus among n processes using objects of type T and registers, and there is no such algorithm for consensus among n+1 processes.

If no such n exists, the consensus number is said to be infinite. (I.e. cons(T) $= \infty$).

Examples.

Cons(FIFO queue) $= 2$
Cons(Read/Write Register) $= 1$
Cons(Compare and Swap) $= \infty$

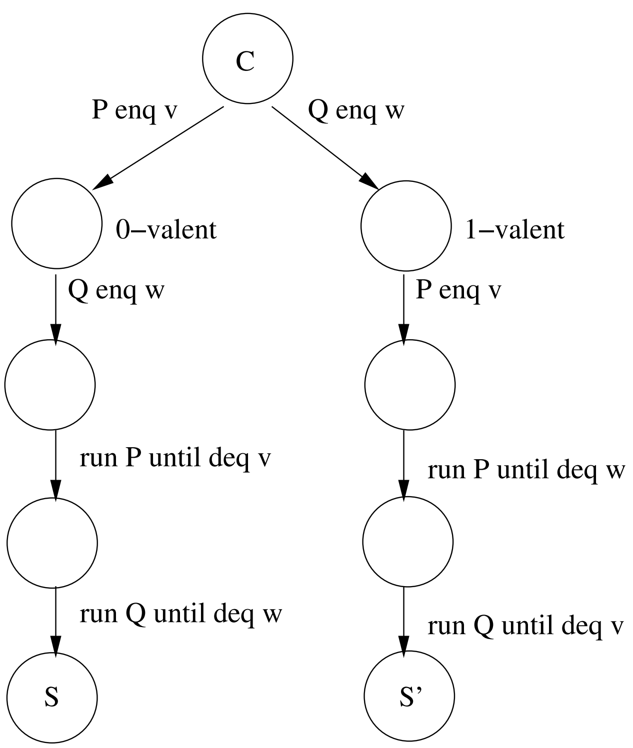

Figure 4: Case IV. Both P and Q execute *enqueue* operations.