Importance of the Consensus Hierarchy

We have seen that the classification of types according to their consensus number tells us a lot about the power of those type. In fact, we have a theorem that says certain implementations are possible and one that says certain implementations are not:

Objects of type T and registers can implement *any* object type in a system of $\text{cons}(T)$ or fewer processes (see Feb 27 lecture).

If $\text{cons}(T_1) > \text{cons}(T_2)$, then T_2 objects and registers cannot implement T_1 objects in a system of more than $\text{cons}(T_1)$ processes (see Feb 25 lecture).

But does the hierarchy tell us everything about the relative power of different types?

No. For example, if $\text{cons}(T_1) > \text{cons}(T_2)$, it doesn't tell us whether T_1 objects (and registers) can implement T_2 objects in a system of more than $\text{cons}(T_1)$ processes.

Robustness of the Consensus Hierarchy

Another interesting question that remains is whether weak types (i.e. types with small consensus numbers) can be used *in combination* to solve hard problems.

Question If $\text{cons}(T_1) < n$ and $\text{cons}(T_2) < n$), can consensus be solved using T_1 and T_2 objects and registers for n processors ?

Yes, sometimes: There are examples of objects that are weak individually but become strong in combination.

Example

$\exists T_1$, T_2 with $\text{cons}(T_1) = 1$ and $\text{cons}(T_1) = 1$ such that $\text{cons}(T_1, T_2) = \infty$. (See Schenk PODC97, and Lo & Hadzilacos SIAM J. Computing 2000.)

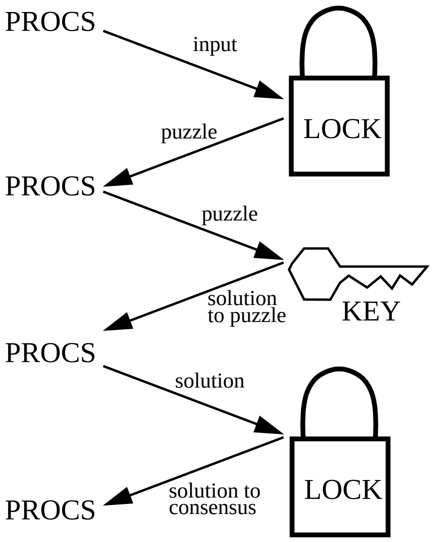

Figure 1: Lock-Key protocol

- T_1 is a Lock
- T_2 is a Key
- Lock object solves consensus but it only reveals answer if the processor has the Key. When processes first access the Lock, it responds with a puzzle. When processes provide the answer to the puzzle, the Lock object reveals the answer to consensus. The Key object solves the puzzle.
- Lock and Key are carefully designed so that they cannot be used by themselves to solve consensus.

The Lock object is *strange*: you wouldn't expect to find an object like it in a real system.
For *nice* objects those funny interactions don't happen and $\text{cons}(T_1, T_2) = \max(\text{cons}(T_1), \text{cons}(T_2))$.
An object is *nice* if, for example, it is *readable* or *RMW*.

Snapshots

Problem With ordinary registers: it's hard to get a consitent view of a set of registers while other processors are concurrently updating them.

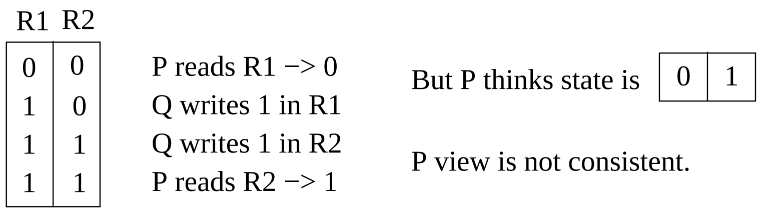

Figure 2: Inconsistent view from set of ordinary registers

A single-writer snapshot object for n processes

State: Vector of n values (each value is a natural number).

Two types of operations:

- *scan*: Output vector stored in state whitout changing it.
- $update(i, v)$ changes component i of vector to value v, $1 \leq i \leq n$ and $v \in N$. $update(i, v)$ can only be performed by process i.

Can we Implement this with registers ? Yes. We'll see how next day.
(Therefore cons(snapshot) = 1.)
Snapshots objects are more convenient to program but real systems don't come equipped with them in hardware.
Fortunately we can build Snapshots objects from registers (in a wait-free manner).
But Snapshots are not as efficient as registers.