SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

CONCURRENT SYSTEMS
LECTURE 3

Prof. Daniele Gorla

MUTEX-free Concurrency

Critical sections (i.e., locks) have drawbacks:

- If not put at the right level of granularity, they unnecessarily reduce concurrency (and efficency)
- Delays of one process may affect the whole system (limit case: crash during a CS)

MUTEX-freedom: the only atomicity is the one provided by the privitives themselves (no CSs)

→ the liveness properties used so far cannot be used anymore, since they rely on CSs

1. **Obstruction freedom**: every time an operation is run in isolation (no overlap with any other operation on the same object), it terminates.
2. **Non-blocking**: whenever an operation is invoked on an object, eventually one operation on that object terminates *(reminds deadlock-freedom in MUTEX-based concurrency)*
3. **Wait freedom**: whenever an operation is invoked on an object, it eventually terminates *(reminds starvation-freedom in MUTEX-based concurrency)*
4. **Bounded wait freedom**: W.F. plus a bound on the number of steps needed to terminate *(reminds bounded bypass in MUTEX-based concurrency)*

REMARK: these notions natually cope with (crash) failures → fail stop is another way of terminating

→ there is no way of distinguishing a failure from an arbitrary long sleep (bec. of asynchrony)

REMARK: we can provide a result like the Round Robin, provided that we have *failure detectors*

A wait-free Splitter

Assume to have atomic R/W registers.

A **splitter** is a concurrent object that provides a single operation dir such that:

1. (*validity*) it returns L, R or S (left, right, stop)
2. (*concurrency*) in case of n simultaneous invocations of dir
   a. At most n-1 L are returned
   b. At most n-1 R are returned
   c. At most 1 S is returned
3. (*wait freedom*) it eventually terminates

Idea:

- Not all processes obtain the same value
- In a solo execution (i.e., without concurrency) the invoking process must stop (0 L && 0 R && at most 1 S)

A wait-free Splitter

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

We have:

- DOOR : MRMW boolean atomic register initialized at 1
- LAST : MRMW atomic register initialized at whatever process index

```
dir(i) :=
     LAST ← i
     if DOOR = 0 then return R
                 else DOOR ← 0
                      if LAST = i then return S
                                  else return L
```

With 2 processes, you can have

- One goes left and one goes right
- One goes left and the other stops
- One goes right and the other stops

An Obstruction-free Timestamp Generator

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

A **timestamp generator** is a concurrent object that provides a single operation get_ts such that:

1. (*validity*) not two invocations of get_ts return the same value
2. (*consistency*) if one process terminates its invocation of get_ts before another one starts, the first receives a timestamp that is smaller than the one received by the second one
3. (*obstruction freedom*) if run in isolation, it eventually terminates

Idea: use something like a splitter for possible timestamp, so that only the process that receives S (if any) can get that timestamp.

An Obstruction-free Timestamp Generator

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

We have:

- DOOR[i] : MRMW boolean atomic register initialized at 1, for all i
- LAST[i] : MRMW atomic register initialized at whatever process index, for all i
- NEXT : integer initialized at 1

```
get_ts(i) :=
     k ← NEXT
     while true do
            LAST[k] ← i
            if DOOR[k] = 1 then
                   DOOR[k] ← 0
                   if LAST[k] = i then NEXT++
                                       return k
            k++
```

Universal Object

Given objects of type T and an object of type Z, is it possible to wait-free implement Z by using only objects of type T and atomic R/W registers?

The **type** of an object is

1. The set of all possible *values for states* of objects of that type
2. A set of *operations* for manipulating the object, each provided with a *specification*, i.e. a description of the conditions under which the operation can be invoked and the effect of the invocation

Here, we focus on types whose operations are

- *Total* : all operations can be invoked in any state of the object
- *Sequentially specified*: given the initial state of an obj, the behaviour depends only by the sequence of operations, where the output to every op. invocation only depends on the input arguments and the invocations preceding it.
  - → formally, $\delta(s, op(args)) = \{\langle s_1, res_1\rangle, \ldots, \langle s_k, res_k\rangle\}$
  - → it is *deterministic* whenever k=1, for every s and every op(args)

An object of type T_U is **universal** if every other object can be wait-free implemented by using only objects of type T_U and atomic R/W registers.

Consensus object

A **consensus object** is a *one-shot object* (i.e., an object such that any process can access it at most once) whose type has only one operation propose(v) such that:

1. (*Validity*) The returned value (also called *the decided value*) is one of the arguments of the propose (i.e., a *proposed value*) in one invocation done by a process (also called a *participant*)
2. (*Integrity*) every process decides at most once
3. (*Agreement*) The decided value is the same for all processes
4. (*Wait-freedom*) every invocation of propose by a correct process terminates

Conceptually, we can implement a consensus object by a register X, initialized at ⊥, for which the propose operation is atomically defined as

```
propose(v)      :=      if X = ⊥ then X ← v
                        return X
```

Universality of consensus holds as follows:

- Given an object O of type Z
- Each participant runs a local copy of O, all initialized at the same value
- Create a total order on the operations on O, by using consensus objects
- Force all processes to follow this order to locally simulate O

  → all local copies are consistent

A wait-free construction (for sequential spec's)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

LAST_OP[1..n] : array of SWMR atomic R/W registers containing pairs init at $\langle \perp, 0 \rangle$ $\forall$i

$last\_sn_i$[1..n] : local array of the last op by pj executed by pi init at 0 $\forall$i,j

op(arg) by pi on Z

```
result_i ← ⊥
LAST_OP[i] ← ⟨op(args),
                last_sn_i[i]+1⟩
wait result_i ≠ ⊥
return result_i
```

local simulation of Z by pi

```
k ← 0
z_i ← Z.init()
while true
   invoc_i ← ε
   ∀ j = 1..n
      if π2(LAST_OP[j])>last_sn_i[j]
      then invoc_i.append(
                  ⟨π1(LAST_OP[j]),j⟩ )
   if invoc_i ≠ ε then
      k++
      exec_i←CONS[k].propose(invoc_i)
      for r=1 to |exec_i|
         ⟨z_i,res⟩ ← δ(z_i,π1(exec_i[r]))
         j ← π2(exec_i[r])
         last_sn_i[j]++
         if i=j then result_i←res
```

Here, π_1 and π_2 denote the projections (i.e., the first and second elements) of a pair

Solution for non-deterministic spec.'s

If the specifications of Z's operations are *non-deterministic*, then δ does not return one single possible pair after one invocation, but a set of possible choices.

How to force every process to run the very same sequence of operations on their local simulations?

1. Brute force solution: for every pair $\langle s , op(args)\rangle$, fix a priori one element of $\delta(\langle s , op(args)\rangle)$ to be chosen
   → «cancelling» non-determinism

2. Additional consensus objects, one for every element of every list

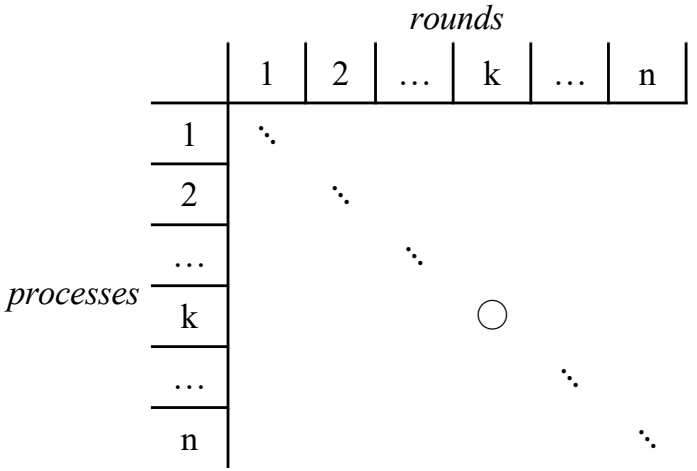
3. Reuse the same consensus object: for all k, CONS[k] not only chooses the list of invocations, but also the final state of every invocation
   → the proposals should also pre-calculate the next state and propose one

Binary vs Multivalued Consensus

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Binary consensus: just 2 possible proposals (say, 0 and 1)

Multivalued consensus (with unbounded values)



IDEA:

- we have n processes and n binary consensus rounds;
- at round k, all processes propose 1 if p_k has proposed something, 0 otherwise.
- If the decided value is 1, then decide p_k's proposal; otherwise, go to the next round.

```
PROP[1..n] array of n proposals, all init at ⊥
BC[1..n] array of n binary consensus objects
mv_propose(i,v) :=
   PROP[i] ← v
   for k=1 to n do
        if PROP[k] ≠ ⊥ then res ← BC[k].propose(1)
                       else res ← BC[k].propose(0)
        if res = 1 then return PROP[k]
   wait forever
```

Binary vs Multivalued Consensus

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Validity, Agreement, Integrity, Termination:

- Let p_x the first process that writes a proposal
- every p_i that participates to the consensus reads the other proposals after that it has written PROP[i]

  → all participants start their for loops after that p_x has written its proposal
- every p_i that participates to the consensus finds PROP[x] ≠ ⊥ in their for loop
- BC[x] only receives proposals equal to 1
- Because of validity of binary consensus, BC[x] returns 1
- every p_i that participates to the consensus receives 1 at most in its x-th iteration of the `for`
- Let y (≤ x) be the first index such that BC[y] returns 1

  → BC[z] = 0 for all z < y
- Since all participating processes invoke the binary consensus in the same order, they all decide the value proposed by p_y and terminate

Binary vs Bounded Multivalued Consensus

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Multivalued consensus (with bounded values)

Let N be the number of possible proposals and h = $\lceil log_2 N \rceil$ be the number of bits needed to binary represent them (this value is known to all processes).

→ IDEA: decide bit by bit the final outcome

```
PROP[1..n][1..h] array of n h-bits proposals, all init at ⊥
BC[1..h] array of h binary consensus objects

bmv_propose(i,v) :=
   PROP[i] ← binary_repr_h(v)
   for k=1 to h do
        P ← {PROP[j][k] | PROP[j]≠⊥ ∧ PROP[j][1..k-1]=res[1..k-1]}
        let b be an element of P
        res[k] ← BC[k].propose(b)
   return value(res)
```

Binary vs Bounded Multivalued Consensus

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

- *Wait freedom*: trivial
- *Integrity*: trivial
- *Agreement*: by agreement of the h binary consensus objects
- *Validity*: for all k, we prove that, if dec is the decided value, then there exists a j such that p_j is partecipating (i.e., PROP[j] $\neq \perp$) and dec[1..k] = PROP[j][1..k]

  → By construction, P contains the k-th bits of the proposals whose first (k-1) bits coincide with the first k-1 bits decided so far:

  for every b $\in$ P, there exists a j such that PROP[j] $\neq \perp$,

  PROP[j][1..k-1] = dec[1..k-1] and PROP[j][k] = b

  → whatever b $\in$ P is selected in the k-th binary consensus, there exists a j such that PROP[j] $\neq \perp$ and PROP[j][1..k] = dec[1..k]

  → by taking k = h, we can conclude.

Consensus Number

Which objects allow for a wait free impementation of (binary) consensus?

→ the answer depends on the number of participants

The **consensus number** of an object of type T is the greatest number n such that it is possible to wait free implement a consensus object in a system of n processes by only using objects of type T and atomic R/W registers.

For all T, CN(T) > 0; if there is no sup, we let CN(T) := +∞

Thm: let CN(T1) < CN(T2), then there exists no wait free implementation of objects of type T2 in a system of n processes that only uses objects of type T1 and atomic RW reg.s, for all n s.t. CN(T1) < n ≤ CN(T2).

Proof

- Fix such an n; by contr., there exists a wait free implementation of objects of type T2 in a system of n processes that only uses objects of type T1 and atomic RW reg.s.
- Since n ≤ CN(T2), by def. of CN, there exists a wait free implementation of consensus in a system of n processes that only uses objects of type T2 and atomic RW reg.s.
- Hence, there exists a wait free implementation of consensus in a system of n processes that only uses objects of type T1 and atomic RW reg.s.

→ contraddiction with CN(T1) < n Q.E.D.

Schedules and Configurations

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Schedule = sequence of operation invocations issued by processes

Configuration = the global state of a system at a given execution time (values of the shared memory + local state of every process)

Given a configuration C and a schedule S, we denote with S(C) the configuration obtained starting from C by applying S

Let us consider binary consensus implemented by an algorithm A.

A configuration C obtained during the execution of A is called

- **v-valent** if S(C) decides v, for every S schedule of A;
- **monovalent**, if there exists $v \in \{0,1\}$ s.t. C is v-valent;
- **bivalent**, otherwise.

Fundamental theorem

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Thm: If A wait-free implements binary consensus for n processes, then there exists a bivalent initial configuration.

Proof:

1. Let C_i be the initial config. where all p_j (for $j \leq i$) propose 1 and all the others propose 0
2. By validity, C_0 is 0-valent and C_n is 1-valent
3. By contradiction, assume all C_i to be monovalent
4. By (2), there exists an i such that C_{i-1} is 0-valent and C_i is 1-valent
5. By definition, C_{i-1} and C_i only differ in the value proposed by p_i (0 and 1, resp.)
6. Consider an execution of A where p_i is blocked for a very long period
    - by wait freedom, all other processes eventually decide
    - call S the scheduling from the beginning to the point in which all processes but p_i have decided
    - since C_{i-1} is 0-valent, all other processes decide 0
    - By (5) and because p_i is sleeping in S, also in $S(C_i)$ all other processes decide 0
    - In $S(C_i)$ we resume p_i and lead it to a decision
        - If it decides 1, we contradict agreement
        - If it decides 0, we contradict 1-valence of C_i.

Q.E.D.

CN(Atomic R/W registers) = 1

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Thm: There exists no wait-free implementation of binary consensus for 2 processes that uses atomic R/W registers.

Proof:

By contradiction, assume A wait-free, with processes p and q.

By ther previous result, it has an initial bivalent configuration C.

→ let S be a sequence of operations s.t. C' = S(C) is maximally bivalent (i.e., p(S(C)) is 0-valent and q(S(C)) is 1-valent, or viceversa)

p(C') can be R1.read() or R1.write(v) and q(C') can be R2.read() or R2.write(v')

1. R1 ≠ R2 

   Whatever operations p and q issue, we have that q(p(C')) = p(q(C'))

   But q(p(C')) is 0-val (because p(C') is) whereas p(q(C')) is 1-val

2. R1 = R2 and both operations are a read 

   Like point (1)

CN(Atomic R/W registers) = 1

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

3. R1 = R2, with p that reads and q that writes (or viceversa) 

   Remark: only p can distinguish C' from p(C') (reads put the value read in a local variable, visible only by the process that performed the read)

   Let S' be the scheduling from C' where p stops and q decides:

   → S' starts with the write of q

   → S' leads q to decide 1, since q(C') is 1-val

   Consider p(C') and apply S'

   → because of the initial remark, q decides 1 also here

   Reactivate p

   → if p decides 0, then we would violate agreement

   → if p decides 1, we contradict 0-valence of p(C')

4. R1 = R2 and both operations are a write 

   Remark: q(p(C)) = q(C) cannot be distinguished by q since the value written by p is lost after the write of q

   Then, work like in case (3).

Q.E.D.

CN(Test&set) = 2

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

```
TS a test&set object init at 0
PROP array of proposals, init at whatever

propose(i, v) :=
      PROP[i] ← v
      if TS.test&set() = 0 then return PROP[i] else return PROP[1-i]
```

Wait-freedom, Validity and Integrity hold by construction.

Agreement: the first that performs test&set receives 0 and decides his proposal; the other one receives 1 and decides the other proposal.

We can then prove the following theorem (the proof is similar in spirit to the previous one), that ensures that CN(Test&set) = 2.

Thm: There exists no wait-free implementation of a binary consensus object for 3 processes that uses atomic R/W registers and test&set objects.

CN(Swap) = CN(Fetch&add) = 2

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

```
S a swap object init at ⊥
PROP array of proposals, init at whatever

propose(i, v) :=
     PROP[i] ← v
     if S.swap(i) = ⊥ then return PROP[i] else return PROP[1-i]
```

```
FA a fetch&add object init at 0
PROP array of proposals, init at whatever

propose(i, v) :=
     PROP[i] ← v
     if FA.fetch&add(1) = 0 then return PROP[i] else return PROP[1-i]
```

REMARK: Similarly to Test&set, we can prove that no consensus is possible with 3 processes.

CN(Compare&swap) = ∞

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Let us consider a verison of the compare&swap where, instead of returning a boolean, it always returns the previous value of the object, i.e.:

```
X.compare&swap(old,new) :=
        atomic { tmp ← X
                 if tmp = old then X ← new
                 return tmp
```

```
CS a compare&swap object init at ⊥

propose(v) :=
     tmp ← CS.compare&swap(⊥,v)
     if tmp = ⊥ then return v else return tmp
```

Remark: also the compare&swap that returns booleans has an infinite CN, but the implementation is less clean (try as an exercise!)

Conclusions

- Concurrency is a powerful tool for enhancing programming
- It adds complications arising from (the necessary) inter-process interactions
- One way of solving these complications is through Mutual Exclusion
  - We can enforce MUTEX in many frameworks, ranging from the vary basic model of Safe Registers, to sophisticated HW primitives (test&set, swap, compare&swap, fetch&add..), by passing through atomic R/W registers
  - We can devise higher-level programming tools (semaphores and monitors) that ensure MUTEX without charging to the programmer the weigth of handling concurrency
- MUTEX is not strictly needed, since we can program concurrent systems even without it
  - This has the advantage of allowing process failues
  - We proved that having just one kind of concurrent object (namely, (binary) consensus objects) is enough for having MUTEX-free concurrency
  - According to the number of processes for which the consensus is feasible, we built a hierarchy of the primitives studied in the framework of MUTEX-based concurrency.