SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

CONCURRENT SYSTEMS
LECTURE 7

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 wrapping of code into 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 natuarlly 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)

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

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

<u>Thm (soundness):</u> this implemenatation satisfies the 3 requirements for the splitter

Proof:

Termination and validity are trivial. For concurrency, we observe that:

1. Not all proc's can obtain R
   → to obtain R, the door must have been closed and who closed the door cannot obtain R
2. Not all proc's can obtain L
   → let us consider the last process that writes into LAST (this is an atomic register, so this is meaningful)
   → if the door is closed, it receives R and √
   otherwise, it finds LAST=i and receives S → √
3. Let pi be the first process that receives S → LAST=i in its second if

```
pi          LAST←i                 LAST=i
     ------------|----------------------|-------------------->
                 \__________ __________/
                            v
              No pj has written into LAST
```

→ it has written LAST before i →it doesn't find LAST=j in its second if and receives L → √

→ it has written LAST after i → it finds the door closed and receives R → √

An Obstruction-free Timestamp Generator

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

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++
```

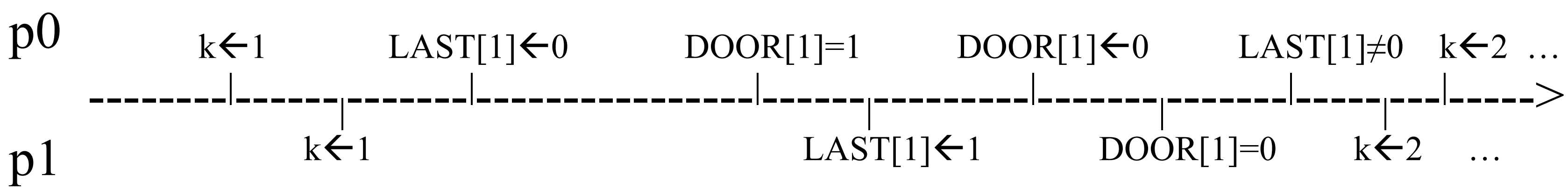
<u>Thm (soundness):</u> this implementation satisfies the 3 properties of the timestamp generator

Proof:

1. Validity holds because of property 2.c of the splitter
2. For consistency, the invocation that terminates increased the val of NEXT before terminating

   → every process that starts after its termination will find NEXT to a greater value (NEXT never decreases!)
3. Obstruction freedom is trivial

REMARK: this implementation doesn’t satisfy the non-blocking property:



A Wait-free Stack

REG is an unbounded array of atomic registers (the stack)

For all i, REG[i] can be

- Written
- Read by the swap(v) primitives (that atomically writes a new value in it)
- Initialized at ⊥

NEXT is an atomic register (pointing at the next free location of the stack) that can be

- Read
- Fetch&add
- Initialized at 1

```
push(v) :=
    i ← NEXT.fetch&add(1)
    REG[i] ← v
```

```
pop() :=
    k ← NEXT-1
    for i=k downto 1
          tmp ← REG[i].swap(⊥)
          if tmp≠⊥ then return tmp
    return ⊥
```

REMARK: crashes do not compromise progress!

PROBLEM: unboundedness of REG is not realistic

A Non-blocking Bounded Stack

Idea: every operation is started by the invoking process and finalized by the next process

STACK[0..k] : an array of registers that can be read or compare&setted

→ STACK[i] is actually a pair ⟨val , seq_numb⟩ initialized at ⟨⊥,0⟩

This is needed for the so called ABA problem with compare&set:

- A typical use of compare&set is

```
tmp ← X
…
if X.compare&set(tmp,v) then …
```

- This is to ensure that the value of X has not changed in the computation
- The problem is that X can be changed twice before the comp&set
- Solution: X is a pair ⟨val , seq_numb⟩, with the constraint that each modification of X increases its seq_numb

  → with the comp&set you mainly test that the seq_numb has not changed

TOP : a register that can be read or compare&setted

→ TOP is actually a triple ⟨index , val , seq_numb⟩ initialized at ⟨0,⊥,1⟩

where the top is in STACK

the pair to be put at the top of STACK

A Non-blocking Bounded Stack


```
push(w) :=
   while true do
        ⟨i,v,s⟩ ← TOP
        conclude(i,v,s)
        if i=k then return FULL
        newtop ← ⟨i+1,w,STACK[i+1].seq_num+1⟩
        if TOP.compare&set(⟨i,v,s⟩,newtop)
        then return OK
```

```
pop() :=
   while true do
        ⟨i,v,s⟩ ← TOP
        conclude(i,v,s)
        if i=0 then return EMPTY
        newtop ← ⟨i-1,STACK[i-1]⟩
        if TOP.compare&set(⟨i,v,s⟩,newtop)
        then return v
```

```
conclude(i,v,s) :=
    tmp ← STACK[i].val
    STACK[i].compare&set(⟨tmp,s-1⟩,⟨v,s⟩)
```

A Non-blocking Bounded Stack

Thm (liveness): the implementation of the stack is non-blocking.

Proof:

Let us consider an operation invocation performed by p

- if it terminates → √
- otherwise, TOP has changed between the first of TOP and the last Compare&set
  - → the only instruction that modifies TOP is the closing Compare&set
  - → another operation invocation (issued by another process) has terminated → √

REMARK: the fact that the operation is concluded by the next process, together with atomicity of compare&set, ensures correctness even with crash failures

→ if it was part of the invocation (just before the final return of push/pop), a failure just after the TOP.compare&set would compromise consistency