An Introduction to the Implementation of Concurrent Objects

Michel RAYNAL

raynal@irisa.fr

Institut Universitaire de France

IRISA, Université de Rennes, France

Summary

- Concurrent objects
- Safety: Linearizability vs sequential consistency
- Lock-based implementations
- Mutex-free implementations
- Liveness: Progress conditions
- Hybrid implementations
- Conclusion

Source

- The content of these slides are from chapters 2, 5 and 6 of the book (composed of 17 chapters)
- Concurrent Programming:
  Algorithms, Principles and Foundations.
  Springer, 515 pages, 2013 (ISBN 978-3-642-32026-2)

Part I

Concurrent Objects

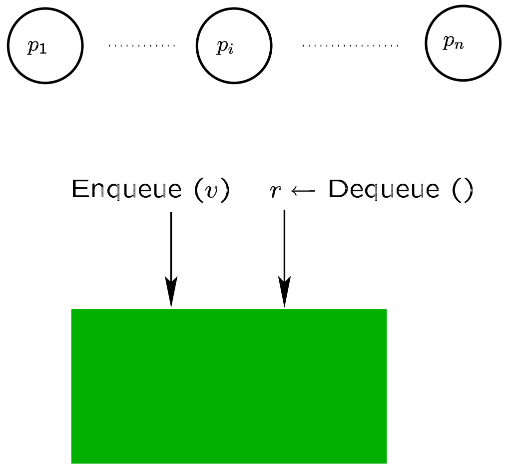
The aim is here to get an intuition of what is a concurrent object

Computation Model

- A set Π of n asynchronous processes $p_1, \ldots, p_n$
- A shared memory made up of atomic read/write registers
- Failure model: process crash model
  - ⋆ Terminology:
    Correct = a process that never crashes
    Faulty = a process that crashes
  - ⋆ $t =$ upper bound on the nb of faulty processes
  - ⋆ Failure-free: $t = 0$
  - ⋆ Wait-free model $t = n - 1$
  - ⋆ t-resilient model: $1 \leq t < n$

Concurrent Object

An object accessed by *concurrent* processes



p_1 ………… p_i ……………… p_n

Enqueue (v) $r \leftarrow$ Dequeue ()

Concurrent Object

- Defined by a *sequential specification*
  - ⋆ Stack, queue, graph, set, etc.
- Defined by a *non-sequential specification*
  - ⋆ Rendezvous object,
  - ⋆ Non-blocking atomic commit (NBAC)

Non seq. specification: the NBAC example

Each process is assumed to vote *yes* or *no*

- Termination. A process that does not crash decides
- Agreement. No two processes decide differently
- Validity. A decided value is *abort* or *commit*
  - ⋆ Justification. *commit* decided ⇒ all processes have voted *yes*
  - ⋆ Obligation. No process crashes and all processes vote *yes* ⇒ *commit* is decided

Object considered here

- Sequential specification
- With *total operations*:

  An operation can always return a result (no blocking imposed by the spec)

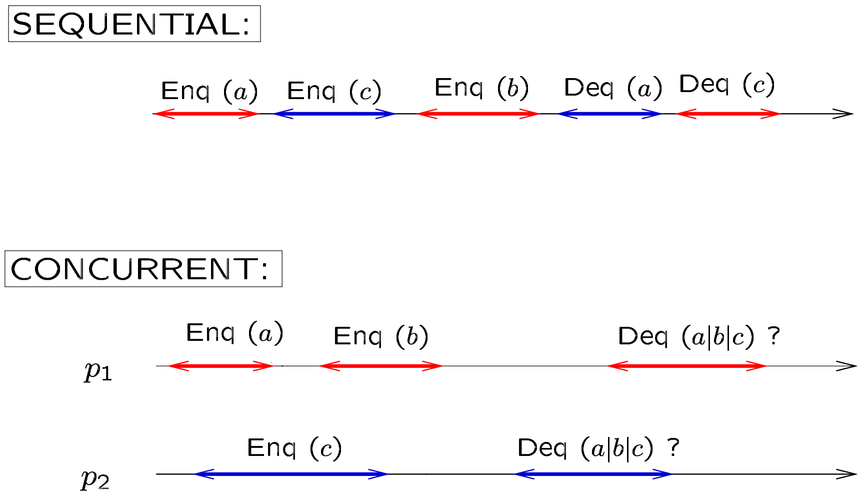
  ⋆ E.g., $pop()$ on an empty stack returns $empty$

  ⋆ E.g., $enqueue()$ on a full bounded queue returns $full$

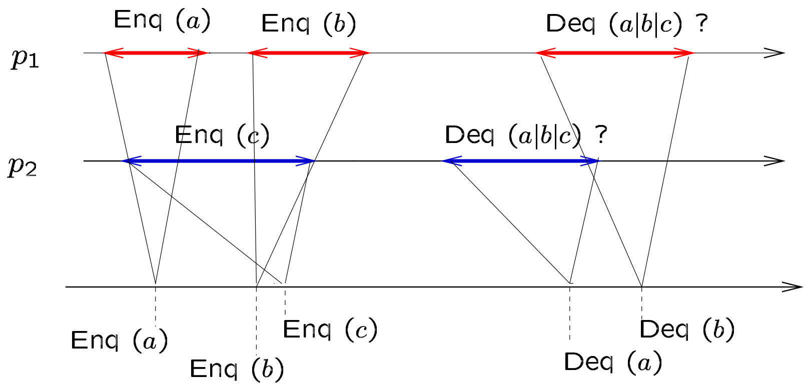
  Hence, (if any) blocking is due to the implementation, not to the spec

---

Sequential vs Concurrent (1)

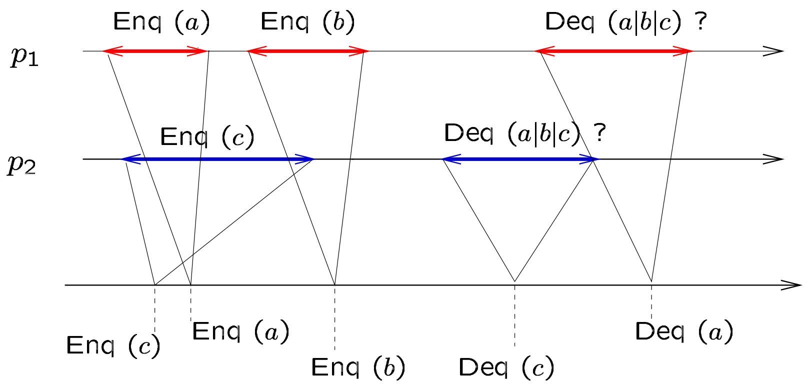


Sequential vs Concurrent (2)



This “history” belongs to the sequential specification

Sequential vs Concurrent (3)



This “history” belongs to the sequential specification

Part II

On the SAFETY side:
Consistency conditions

The aim is here to answer the question:

what is a correct execution involving a set of objects?

Linearizability

- a history is *linearizable* if
  - ⋆ each operation appears as if it has been executed instantaneously at some point of the time line between its start event and its end event
  - ⋆ no two operations appear at the same point of the time line
  - ⋆ the corresponding sequence belongs to the specification of the objects

- Herlihy M.P. and Wing J.M., Linearizability: a correctness condition for concurrent objects. *ACM Toplas*, 12(3):463-492, 1990

Atomicity vs Linearizability

- Atomicity first introduced for read/write registers

  - Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85, 1986

  -Lamport L., On interprocess communication, Part II: algorithms. *Distributed Computing*, 1(2):77-101, 1986

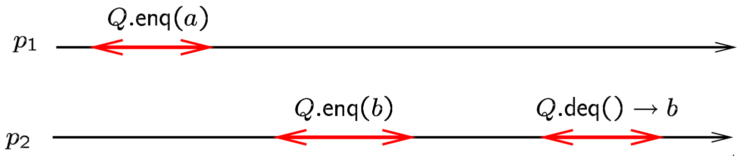
- Linearizability extends Atomicity to any object with a sequential specification

- Hence, Atomicity and Linearizability can be considered as synonymous

---

Another consistency condition: seq consistency

- Similar to Linearizability without requiring agreement with real time



- Lamport L., How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690-691, 1979

Seq consistency is more interesting in message-passing systems

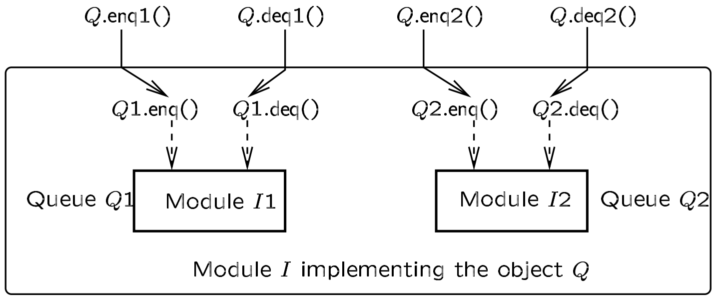
The fundamental difference: composability

- Locality property: A property P is *local* if a set of objects as a whole satisfies P whenever each object satisfies P
- Locality = modularity

  independent implementations compose for free

- Linearizability is a local property
- Sequential consistency is a not local property

The benefit of linearizability



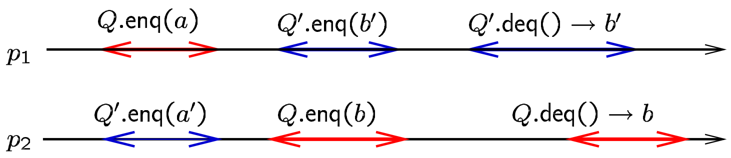
Q.enq1() Q.deq1() Q.enq2() Q.deq2()

$Q1$.enq() $Q1$.deq() $Q2$.enq() $Q2$.deq()

Queue $Q1$ Module $I1$ Module $I2$ Queue $Q2$

Module I implementing the object Q

Seq consistency is not a local property



p_1 Q.enq(a) Q'.enq(b') Q'.deq() $\rightarrow b'$

p_2 Q'.enq(a') Q.enq(b) Q.deq() $\rightarrow b$

Part III

Lock-based Implementations

Classical approaches

- Lock = Mutual exclusion
- Lock from read/write registers
- Low level locks: Semaphores
- Imperative language: monitors (Hoare, Brinch Hansen)
- Declarative language: path expressions (Campbell)

On the liveness side: liveness conditions

- Deadlock-freedom:
  At least one operation invocation always terminates
- Starvation-freedom:
  All operation invocations terminate

From deadlock-free lock to starvation-free lock

Such a construction is based on

- An SWMR array $FLAG[1..n]$ with an entry per process (init to $[down, .., down]$)
- A MWMR register $TURN$ which contains a proc identity
- A deadlock-free lock DF_LOCK (e.g., Lamport's fast mutex algorithm)

- Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, 2006 (ISBN 0-131-97259-6)

- Lamport L., Fast mutual exclusion. *ACM TOCS*, 5(1):1-11, 1987

The construction

```
operation acquire_SF_lock($i$) is
    $FLAG[i] \leftarrow up$;
    wait $[(TURN = i) \lor (FLAG[TURN] = down)]$;
    $DF\_LOCK$.acquire_DF_lock($i$);
    return()
end operation.

operation release_DF_mutex($i$) is
    $FLAG[i] \leftarrow down$;
    if $(FLAG[TURN] = down)$
       then $TURN \leftarrow (TURN \bmod n) + 1$
    end if;
    $DF\_LOCK$.release_DF_lock($i$);
    return()
end operation.
```

Reminder

From a computability point of view

- Mutex can be implemented in crash-free systems from atomic read/write registers
- b-valued atomic read/write registers can be built from safe bits
- Mutex can be implemented directly from safe registers

- Lamport L., A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453-455, 1974

- Aravind A.A., Yet another simple solution to the concurrent programming control problem. *IEEE Trans. on Parallel and Distributed Systems*, 22(6):1056-1063, 2011

Part IV

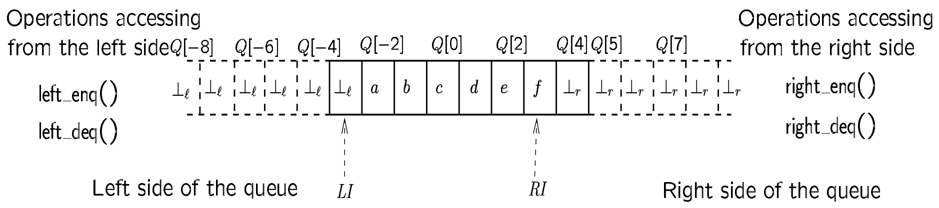
Mutex-free Implementations

Drawbacks of lock-based implementations

- In a lock-based solution: one process at a time can access a given object
- Make the progress of processes depends the ones from the others
  - ⋆ Deadlock-prone
  - ⋆ Cannot cope with the net effect of
    - ∗ asynchrony
    - ∗ and failures
  - ⋆ Process scheduling, swapping

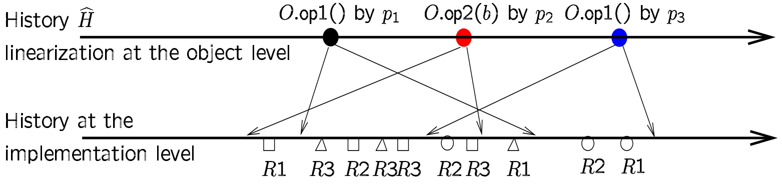
Drawback due lock granularity

Example of a double-ended queue



Mutex-free implementation

Do not use lock (implicitly or explicitly)



No code is protected by a critical section (lock)

- Lamport L., Concurrent Reading and Writing. *CACM*, 20(11):806-811, 1977
- Peterson G.L., Concurrent reading while writing. *ACM TOPLAS*, 5:46-55, 1983
- Herlihy M.P., Wait-free synchronization. *ACM TOPLAS*, 13(1):124-149, 1991

Progress (liveness) conditions

- Obstruction-freedom (is wrt concurrency)
- Non-blocking ($\simeq$ deadlock-freedom)
- Wait-freedom ($\simeq$ starvation-freedom)
  - ⋆ Finite wait-freedom
  - ⋆ Bounded wait-freedom

These progress conditions cope naturally with any asynchrony and crash pattern (while –lock-based– deadlock-freedom and starvation-freedom do not), i.e., they implicitly consider $t = n - 1$ (wait-free model)

Liveness conditions: Summary

Lock-based implementation	Mutex-free implementation
	Obstruction-freedom
Deadlock-freedom	Non-blocking
Starvation-freedom	Wait-freedom

A simple theorem

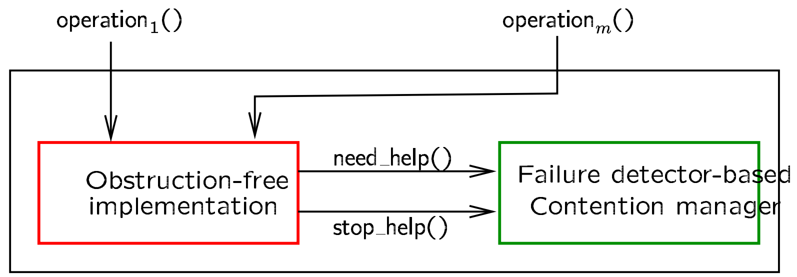
- Context:
  - ⋆ One-shot objects
  - ⋆ Bounded nb of processes
- Theorem: Non-blocking = Wait-free

Boosting obstruction-freedom

- From Obstruction-freedom to non-blocking
- From Obstruction-freedom to wait-freedom
- Failure detector-based contention managers

- Guerraoui R., Kapalka M. and Kuznetsov P., The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6): 415-433, 2008

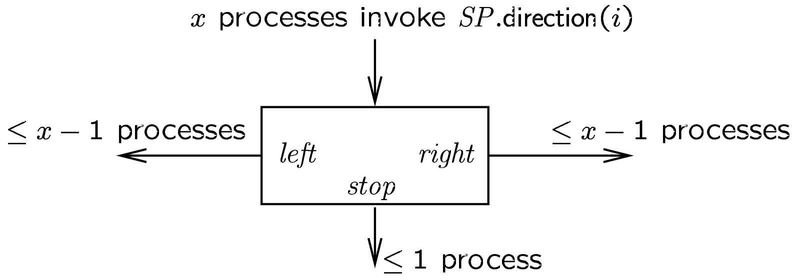
Boosting obstruction-freedom (2)


operation$_1$()
operation$_m$()
Obstruction-free implementation
need_help()
stop_help()
Failure detector-based Contention manager

A very simple wait-free object: the Splitter (1)

- Validity. Value returned by direction() is *right*, *left*, or *stop*
- Concurrent execution. If x processes invoke direction():
  - ⋆ At most $x-1$ processes obtain the value *right*
  - ⋆ At most $x-1$ processes obtain the value *left*
  - ⋆ At most one process obtains the value *stop*
- Termination. Any invocation of direction() terminates

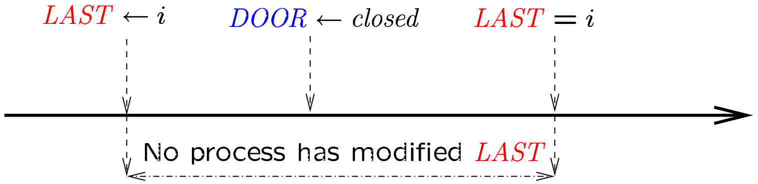
A very simple wait-free object: the Splitter (2)


x processes invoke SP.direction(i)
$\leq x-1$ processes
left
right
stop
$\leq x-1$ processes
≤ 1 process

A very simple wait-free object: the Splitter (3)

```
operation $SP$.direction($i$) is
    $LAST \leftarrow i$;
    if ($DOOR = closed$)
       then return($right$)
       else  ($DOOR \leftarrow closed$;
             if ($LAST = i$)
                then return($stop$)
                else  return($left$)
             end if
    end if
end operation.
```

A very simple wait-free object: the Splitter (4)



Obstruction-free counter (1)

Weak timestamp generator which provides processes with a single operation denoted get_timestamp() which returns an natural integer

- Validity. No two invocations of get_timestamp() return the same value
- Consistency. Let $\mathsf{gt}_1()$ and $\mathsf{gt}_2()$ be two distinct invocations of get_timestamp(). If $\mathsf{gt}_1()$ returns before $\mathsf{gt}_2()$ starts, the timestamp returned by $\mathsf{gt}_2()$ is greater than the one returned by $\mathsf{gt}_1()$
- Termination. Obstruction-freedom

Obstruction-free counter (2)

- $NEXT$: value of the next timestamp, initialized to 1
- $LAST$: unbounded array of atomic registers

  A process p_i deposits its index i in $LAST[k]$ to indicate it is trying to obtain the timestamp k
- $COMP$: unbounded array of atomic Boolean registers initialized to $false$

  A process p_i sets $COMP[k]$ to $true$ to indicate that it is competing for the timestamp k

Obstruction-free counter (2)

```
operation get_timestamp($i$) is
    $k \leftarrow NEXT$;
    repeat forever
        $LAST[k] \leftarrow i$;
        if $(\neg COMP[k])$
            then $COMP[k] \leftarrow true$;
                if $(LAST[k] = i)$
                    then $NEXT \leftarrow NEXT + 1$; return($k$)
                end if
        end if;
        $k \leftarrow k + 1$
    end repeat
end operation.
```

How do processes communicate?

Shared memory models

- Base read/write model
- Base read/write model enriched with specific operations
  - ⋆ Swap (local/shared), Test&Set, Fetch&Add, etc.
  - ⋆ Compare&Swap, LL/SC, etc.
  - ⋆ Herlihy's Hierarchy on the synchro power of base operations define a hierarchy of shared memory models

---

Compare&Swap: definition

```
$X$.compare&swap($old, new$) is
    if $(X = old)$
        then $X \leftarrow new$; return($true$)
        else return($false$)
    end if.
```

Using Compare&Swap

```
statements;
$old \leftarrow X$;
any sequence of statements possibly
involving accesses to the shared memory;
if $X$.compare&swap($old, new$)
    then statements S1
    else statements S2
end if;
statements.
```

Compare&Swap: the ABA problem

- Initially $X = a$
- At time τ_1: p_i reads a from X
- At time $\tau_2 > \tau_1$:
  p_j successfully executes X.C&S(a, b) $(X = b)$
- At time $\tau_3 > \tau_2$:
  p_j successfully executes X.C&S(b, a) $(X = a)$
- At time $\tau_4 > \tau_3$:
  p_i successfully executes X.C&S(a, b) and erroneously believes that X has not been modified by another process in the interval $[\tau_1..\tau_4]$

Solving the ABA problem

Associate a new sequence number with every X.C&S

- X is now a pair $\langle a, sn\rangle$
- At time τ_1:
  p_i reads $\langle a, sn\rangle$ from X
- At time $\tau_2 > \tau_1$:
  p_j successfully executes X.C&S$(\langle a, sn\rangle, \langle b, sn + 1\rangle)$
- At time $\tau_3 > \tau_2$:
  p_k successfully executes X.C&S$(\langle b, sn + 1\rangle, \langle a, sn + 2\rangle)$
- At time $\tau_4 > \tau_3$:
  when p_i executes X.C&S$(\langle a, sn\rangle, \langle c, sn + 1\rangle)$, the write into X fails and returns $false$ to p_i

---

Non-blocking objects based on Compare&Swap

- Non-Blocking Queue Based on Read/Write Registers and Compare&Swap:

  \- Michael M.M. and Scott M.L., Simple, fast and practical blocking and non-blocking concurrent queue algorithms. *Proc. 15th Int'l ACM Symposium on Principles of Distributed Computing (PODC'96)*, ACM Press, pp. 267-275, 1996

  This implementation was included in the standard Java Concurrency Package

- Non-Blocking Stack Based on Compare&Swap Registers

  \- Shafiei N., Non-blocking array-based algorithms for stacks and queues. *Proc. 11th Int'l Conference on Distributed Computing and Networking (ICDCN'09)*, Springer Verlag, LNCS #5408, pp. 55-66, 2009

Uniform presentation of the previous objects and other objects in *Concurrent Programming: Algorithms, Principles and Foundations*, *Springer*, 515 pages, 2013

A wait-free stack (1)

- Based on Fetch&Add and Swap operations
- Uses:
  - ⋆ $REG[0..\infty)$: array of atomic registers which contains the elements of the stack.
    $REG[0]$ contains always the value $\perp$ (used only to simplify the description of the algorithm)
  - ⋆ $NEXT$: atomic register containing the index of the next entry where a value can be deposited, initialized to 1

\- Afek Y., Gafni E. and Morisson A., Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239-252, 2007

A wait-free stack (2)

```
operation push($v$) is
    $in \leftarrow NEXT$.fetch&add() $- 1$;
    $REG[in] \leftarrow v$;
    return()
end operation.

operation $Q$.pop() is
    $last \leftarrow NEXT - 1$;
    for $x$ from $last$ to 0 do
        $aux \leftarrow REG[x]$.swap($\perp$);
        if ($aux \neq \perp$) then return($aux$) end if
    end for,
    return($empty$)
end operation.
```

Part V

Hybrid Implementations

The aim is here to design object implementations merging locks and mutex-freedom

---

Types of hybrid implementations

- Static hybrid
  - ⋆ Some operation implementations are wait-free, other are lock-based
  - ⋆ Example: a concurrent set
- Dynamic hybrid (context sensitive)
  - ⋆ Define a notion of favorable circumstances (wrt failures, concurrency, etc.)
  - ⋆ And the implementation of the operations must not use locks in favorable circumstances

---

Static hybrid set

- Operations
  - ⋆ S.add(v) adds v to the set S and returns *true* if v was not in the set; Otherwise it returns *false*
  - ⋆ S.remove(v) suppresses v from S and returns *true* if v was in the set; Otherwise it returns *false*
  - ⋆ S.contain(v) returns *true* if $v \in S$ and *false* otherwise
- Static hybridism
  - ⋆ S.add() and S.remove(): lock-based but deadlock-free
  - ⋆ S.contain():mutex-free and wait-free

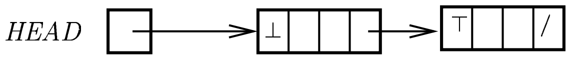
- Heller S., Herlihy M.P., Luchangco V., Moir M., Scherer W.III and Shavit N., A lazy concurrent list-based algorithm. *Parallel Processing Letters*, 17(4):411-424, 2007.

internal Representation

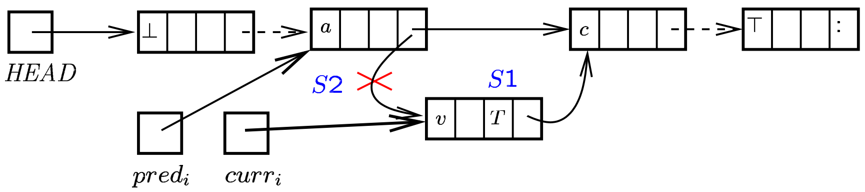
- linked list pointed to by *HEAD*
- A cell of the list (say *NEW_CELL*) is made up of
  -
  - *NEW_CELL.val* which contains a value (element of the set).
  - *NEW_CELL.out*: Boolean set to *true* when the corresponding element is suppressed from the list
  - *NEW_CELL.lock*: lock used to ensure mutual exclusion (when needed) on the cell
  - *NEW_CELL.next*: pointer to the next cell.

Initial state

- The set is organized as a sorted linked list
- All operation algorithms are based on list traversal



Operation S.remove(v): behavior



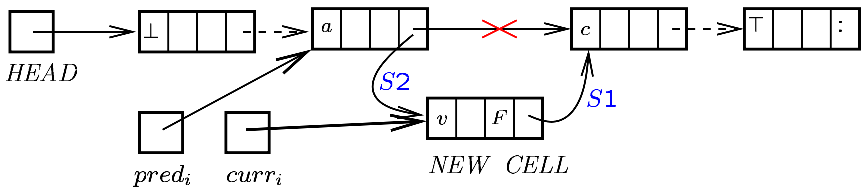
Operation S.remove(v): algorithm

```
operation $S$.remove($v$) is
  $pred \leftarrow HEAD$; $curr \leftarrow (HEAD \downarrow).next$;
  while $((curr \downarrow).val < v)$
     do $pred \leftarrow curr$; $curr \leftarrow (curr \downarrow).next$ end while;
  $((pred \downarrow).lock)$.acquire_lock(); $((curr \downarrow).lock)$.acquire_lock();
  $valid \leftarrow false$;
  if validate$(pred, curr)$
     then $valid \leftarrow true$; $pres \leftarrow ((curr \downarrow).val = v)$;
          if $(pres)$ then $(curr \downarrow).out \leftarrow true$;
                           $(pred \downarrow).next \leftarrow (curr \downarrow).next$
          end if
  end if;
  $((pred \downarrow).lock)$.release_lock(); $((curr \downarrow).lock)$.release_lock();
  if $(valid)$ then return$(pres)$ else restart the operation end if
end operation.
```

Validation predicate

```
internal predicate validate($pred, curr$) is
  let $res = ( \neg ((pred\downarrow).out)$
              $\wedge \neg ((curr\downarrow).out)$
              $\wedge ((pred\downarrow).next = curr))$;
  return($res$)
end internal predicate.
```

Operation S.add(v): behavior



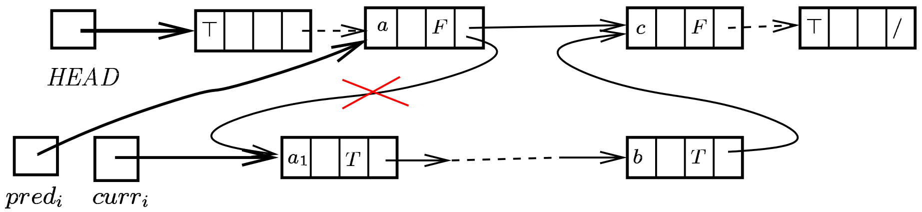
Operation S.add(v): algorithm

```
operation $S$.add($v$) is
  $pred \leftarrow HEAD$; $curr \leftarrow (HEAD\downarrow).next$;
  while $((curr\downarrow).val < v)$
          do $pred \leftarrow curr$; $curr \leftarrow (curr\downarrow).next$ end while;
  $((pred\downarrow).lock)$.acquire_lock(); $valid \leftarrow false$;
  if validate($pred, curr$)
          then $valid \leftarrow true$; $to\_add \leftarrow ((curr\downarrow).val \neq v)$;
               if ($to\_add$) then $S$.add_new_cell() end if
  end if;
  $((pred\downarrow).lock)$.release_lock();
  if ($valid$) then return($to\_add$) else restart the operation end if
end operation.
```

Internal operation S.add_new_cell(): algorithm

```
internal operation $S$.add_new_cell() is
     $NEW\_CELL \leftarrow$ new_cell();
     $NEW\_CELL.out \leftarrow false$;
     $NEW\_CELL.val \leftarrow v$;
     $NEW\_CELL.next \leftarrow curr$;
     $NEW\_CELL.lock \leftarrow open$;
     $(pred\downarrow).next \leftarrow (\uparrow new\_cell)$
end internal operation.
```

Operation S.contain(v): behavior



Operation S.contain(v): algorithm

```
operation $S$.contain$(v)$ is
  $curr \leftarrow HEAD$;
  while $((curr\downarrow).val < v)$ do $curr \leftarrow (curr\downarrow).next$ end while;
  let $res = ((curr\downarrow).val = v) \wedge (\neg(curr\downarrow).out)$;
return$(res)$ end operation.
```

A dynamic hybrid consensus object

- Consensus object
  - ⋆ Validity. A decided value is a proposed value
  - ⋆ Agreement. No two processes decide different values
  - ⋆ Termination. Any invocation of propose() terminates
  - ⋆ Binary consensus: only 0 and 1 can be proposed
- Favorable circumstances: when there is no concurrency or the participating processes propose the same value

- Taubenfeld G., Contention-sensitive data structure and algorithms. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer Verlag, LNCS #5805, pp. 157-171, 2009

Underlying implementation objects

- $PROPOSED[0..1]$, which is an array of two Boolean registers, both initialized to *false*. The atomic register $PROPOSED[v]$ is set to *true* to indicate that a process has proposed value v.
- $DECIDED$: atomic register whose domain is $\{\bot, 0, 1\}$. Initialized to $\bot$, it eventually contains the value that is decided (and never the value which is not decided)
- AUX: atomic register whose domain and initial value are the same as for $DECIDED$
- $LOCK$: starvation-free lock

Dynamic hybrid implementation of binary consensus

```
operation $C$.propose($v$) is
  $PROPOSED[v] \leftarrow true$; if $(AUX = \perp)$ then $AUX \leftarrow v$ end if;
  if $(\neg PROPOSED[1-v])$
     then $DECIDED \leftarrow v$
     else  if $(DECIDED = \perp)$
              then $LOCK$.acquire_lock();
                   if $(DECIDED = \perp)$
                      then $DECIDED \leftarrow AUX$
                   end if;
                   $LOCK$.release_lock()
           end if;
  end if;
  return($DECIDED$)
end operation.
```

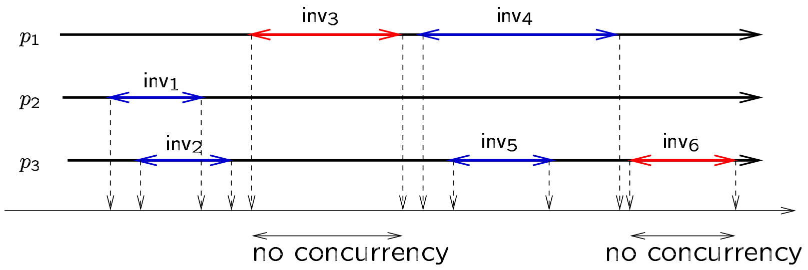
Part VI

Abortable objects

Concurrency abortable object

- Any invocation of an object operation
  - ⋆ Returns after a bounded number of steps (shared memory accesses) and
  - ⋆ is allowed to return the default value $\perp$ in presence of concurrency (then the object has not been modified)
- Can be generalized: An operation is allowed to return $\perp$ only in “unfavorable circumstances”

Illustrating space-time diagram



A non-blocking abortable bounded stack (1)

- The stack is of size k
- Operation push(v)
  - ⋆ returns *full* if the stack is full, otherwise
  - ⋆ adds v to the top of the stack and returns *done*
- Operation pop()
  - ⋆ returns *empty* if the stack is empty, otherwise
  - ⋆ suppresses the value from the top of the stack and returns it

A non-blocking abortable bounded stack (1)

In presence of concurrency

- Operation invocations may return $\perp$ (abortable object)
- But at least one returns a non-$\perp$ value (non-blocking)

Stack representation (1)

- An array $STACK[0..k]$ of atomic registers
- $\forall x: \ 0 \leq x \leq k: \ STACK[x]$ has two fields
  - ⋆ $STACK[x].val$ contains a value
  - ⋆ $STACK[x].sn$ contains a seq number (used to prevent the ABA problem on this register)

    It counts the nb of successful writes on $STACK[x]$

  $\forall x: \ 1 \leq x \leq k: \ STACK[x]$ initialized to $\langle \perp, 0 \rangle$
- $STACK[0]$ always stores a dummy entry (init to $\langle \perp, -1 \rangle$)

Stack representation (2)

- A register TOP that contains the index of the top of the stack plus the corresponding pair $\langle v, sn \rangle$
- TOP initialized to $\langle 0, \perp, 0 \rangle$
- Both $STACK[x]$ and TOP are modified with Compare&Swap

Principle: laziness + helping mechanism

- A push or pop operation
  - ⋆ updates TOP, and
  - ⋆ leaves to the next operation the corresponding update of the stack

    Hence it helps the previous (push or pop) operation by modifying the stack accordingly

Shafiei N.,
Non-blocking Array-based Algorithms for Stacks and Queues.
Proc. th Int'l Conference on Distributed Computing and Networking (ICDCN'09),
Springer Verlag LNCS #5408, pp. 55-66, 2009

Abortable push: weak_push()

```
operation weak_push($v$):
    $(index, value, seqnb) \leftarrow TOP$;
    help($index, value, seqnb$);
    if ($index = k$) then return($full$) end if;
    $sn\_of\_next \leftarrow STACK[index + 1].sn$;
    $newtop \leftarrow \langle index + 1, v, sn\_of\_next + 1\rangle$;
    if $TOP$.C&S($\langle index, value, seqnb\rangle, newtop$)
        then return($done$) else return($\perp$) end if.
```

Abortable stack: help procedure

```
procedure help($index, value, seqnb$):
    $stacktop \leftarrow STACK[index].val$;
    $STACK[index]$.C&S($\langle stacktop, seqnb - 1\rangle, \langle value, seqnb\rangle$).
```

Abortable pop: weak_pop()

```
operation weak_pop():
    $(index, value, seqnb) \leftarrow TOP$;
    help($index, value, seqnb$);
    if ($index = 0$) then return($empty$) end if;
    $belowtop \leftarrow STACK[index - 1]$;
    $newtop \leftarrow \langle index - 1, belowtop.val, belowtop.sn + 1\rangle$;
    if $TOP$.C&S($\langle index, value, seqnb\rangle, newtop$)
        then return($value$) else return($\perp$) end if.
```

From an abortable to a non-blocking stack

```
operation non_blocking_push($v$):
        repeat $res \leftarrow$ weak_push($v$) until $res \neq \perp$ end repeat;
        return($res$).

operation non_blocking_pop():
        repeat $res \leftarrow$ weak_pop() until $res \neq \perp$ end repeat;
        return($res$).
```

From Non-blocking abortable to Starvation-freedom (1)

- Object operations: denoted ABO.ab_oper(par)
- $CONTENTION$: atomic Boolean read/write register, initialized to $false$.

  Used to indicate that there is a process that has acquired the lock and is invoking ABO.ab_oper()
- $LOCK$: a starvation-free lock

From Non-blocking abortable to Starvation-freedom (2)

```
operation oper($par$) is
   if ($\neg CONTENTION$)
      then $res \leftarrow ABO$.ab_oper($par$);
           if ($res \neq \perp$) then return($res$) end if
   end if;
   $LOCK$.acquire_SF_lock();
     $CONTENTION \leftarrow true$;
     repeat $res \leftarrow ABO$.ab_oper($par$) until $res \neq \perp$ end repeat;
     $CONTENTION \leftarrow false$;
   $LOCK$.release_SF_lock();
   return($res$)
end operation.
```

Part VII

Conclusion

What do we have visited?

- Concurrent objects
- Different types of objects
- Safety vs liveness
- Lock-based vs mutex-free implementations
- Notion of a hybrid implementation
- Abortable objects
- Systematic transformations