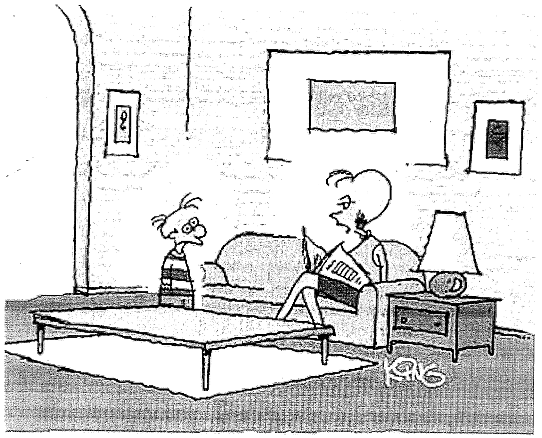
Never forget ....



"No, you weren't downloaded. Your were born."

Concurrent Systems:

Hybrid Object Implementations and Abortable Objects

Michel RAYNAL

raynal@irisa.fr

Institut Universitaire de France

IRISA, Université de Rennes, France

Hong Kong Polytechnic University (Poly U)

The world is changing

- Concurrency in multiprocessors is true concurrency
- It follows that the concurrency concepts and techniques used to cope with multiplexing or interrupt handling are no longer appropriate, and must be revisited to address the current computing world
- "Changes in technology can have far-reaching effects on theory.[...] After decades of being respected but not taken seriously, research on multiprocessor algorithms and data structures is going mainstream"

  Herlihy M.P. and Luchangco V., Distributed computing and the multicore revolution. *ACM SIGACT News*, 39(1): 62-72, 2008

The world is changing (cont'd)

- Before the introduction of multicore processors, parallelism was largely dedicated to computational problems with regular, slow-changing (or even static) communication and coordination patterns. Such problems arise in scientific computing or in graphics, but rarely in systems.

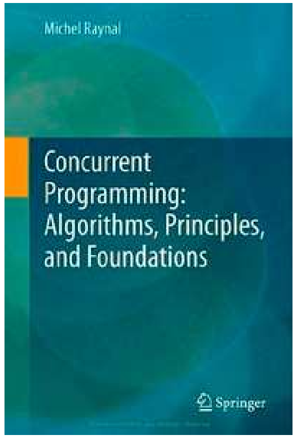
  The future promises us multiple cores on anything from phones to laptops, desktops, and servers, and therefore a plethora of applications characterized by complex, fast-changing ineractions and data exchanges."
- Nir Shavit, Data structures in the multicore age, *Communcations of the ACM*, 54(3):76-84, 2011

Summary

- Objects and Concurrent objects
- On the safety side
- Lock-based implementations
- Mutex-free implementations
- Hybrid implementations
- Abortable objects
- Conclusion

Source

Parts of these slides are inspired from chapters 2, 5, 6, and 8 of the follwing book (composed of 17 chapters)



Concurrent Programming: Algorithms, Principles and Foundations

by Michel Raynal

Springer, 531 pages, 2013

ISBN: 978-3-642-32026-2

Part I

Objects and
Concurrent Objects

Once upon a time ...

- Sequential programming:
  - ⋆ C.A.R. Hoare: the notion of a Record class (1965)
  - ⋆ O.-J., Dahl, K. Nygaard: SIMULA 67 introduced
    - ∗ The notion of an object (encapsulation, prefix/heritage)
    - ∗ The notion of a co-routine (thread)
- Concurrent programming: E.W. Dijkstra (1965)
  - ⋆ Notion of a semaphore, notion of a process
- O.-J. Dahl, E.W.D. Dijkstra et C.A.R. Hoare,
  Structured Programming
  Academic Press, 1972 (ISBN 0-12-200550-3)

Base Computation Model

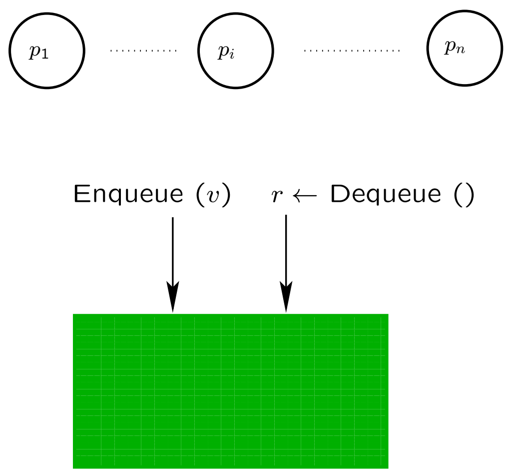
- A set Π of n processes $p_1, \dots, p_n$
  - ⋆ Sequential and asynchronous
- A shared memory: atomic read/write registers
- Failure model: process crash model
  - ⋆ Terminology: given a run
    Correct = a process that does not crash
    Faulty = a process that crashes
  - ⋆ t = max nb of faulty processes
  - ⋆ Failure-free: $t = 0$
  - ⋆ t-resilient model: $1 \leq t < n$
  - ⋆ Wait-free model $t = n - 1$
- Notation $\mathcal{ARW}_{n,t}[\emptyset]$ ($\mathcal{ARW}_n[\emptyset]$ when $t = 0$)

Enriched model

- Base model: communication RW registers only
- Enriched model:
  Stronger communication (synchr) operations
  - ⋆ The meaning of "stronger" is here wrt the computational power in the presence of failures
  - ⋆ Herlihy's hierarchy (consensus numbers)
    - ∗ CN = 1: atomic read/write registers
    - ∗ CN = 2: Test&set, Swap, Fetch&add, ...
    - ∗ $3 \leq$ CN $< +\infty$: ....
    - ∗ CN = $+\infty$: Compare&Swap, LL/SC, Sticky bit, mem-to-mem swap, augmented queue, etc.

Concurrent Object

An object accessed by *concurrent* processes



Concurrent Object (1)

- The locus where processes meet and exchange
  - ⋆ Safety: nothing bad happens
  - ⋆ Liveness: something happens
- The adversaries
  - ⋆ Concurrency
  - ⋆ Non-determinism
  - ⋆ Failures (impact the computability side)

The aim of synchronization is to preserve invariants

- Amdhal's Law ⇒
  the design of concurrent objects is crucial for efficiency

Concurrent Object (2)

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

Concurrent Object with sequential specification

- Easier to understand and prove properties!
- An implementation consists then in mapping concurrent executions (to be efficient) on sequential ones (to reason)

  a parallel execution has to appear as a sequential one

Cope with adversaries

- Safety: weakened semantics
- Liveness: families of progress conditions
  - ⋆ Failure-free system:
    deadlock-freedom, starvation-freedom
  - ⋆ Failure-prone system:
    obstruction-freedom, non-blocking, wait-freedom

One-shot vs multi-shot object

- One-shot:
  a process can invoke an object operation at most once (e.g., consensus, NBAC)
- Multi-shot:
  no restriction on the numer of times a process can invoke an operation (e.g., stack, queue)

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?

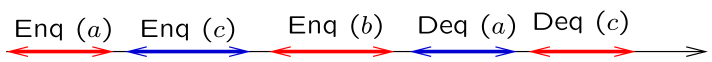
Linearizability

- 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

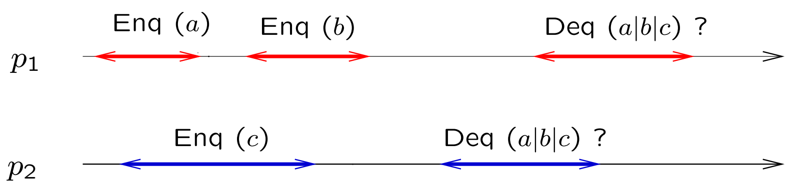
- Herlihy M.P. and Wing J.M., Linearizability: a correctness condition for concurrent objects. *ACM Toplas*, 12(3):463-492, 1990

Sequential vs Concurrent (1)

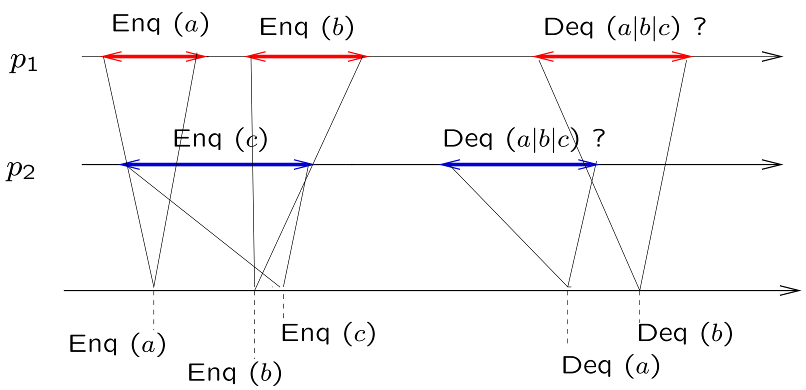
SEQUENTIAL:



CONCURRENT:



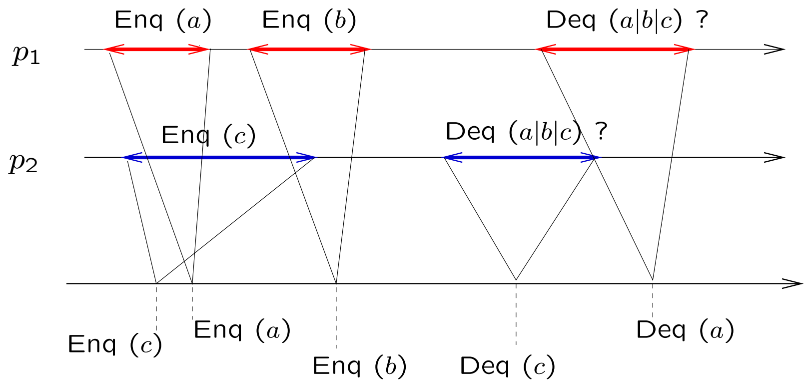
Sequential vs Concurrent (2)



This "history" belongs to the sequential specification

Concurrency $\Rightarrow$ non-determinism

Sequential vs Concurrent (3)



This "history" belongs to the sequential specification

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

The fundamental property: composability

- Linearizability is composable

  An execution is linearizable ⇔
  each of its objects is linearizable

- Locality = modularity

  independent implementations compose for free

- Sequential consistency is a not composable property

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: object's point of view
  At least one operation invocation always terminates
- Starvation-freedom: user's point of view
  All operation invocations terminate

Part IV

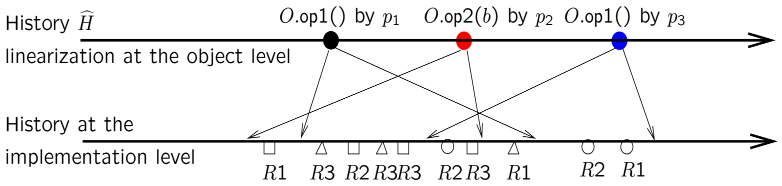
Mutex-free Implementations

Drawbacks of lock-based implementations

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

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 i.e., they implicitly consider $t = n - 1$ (wait-free model)

(lock-based) deadlock/starvation-freedom do not

Non-blocking objects based on Compare&Swap

- Non-blocking queue based on read/write registers and Compare&swap: model $\mathcal{ARW}_{n,n-1}$[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

Liveness conditions: Summary

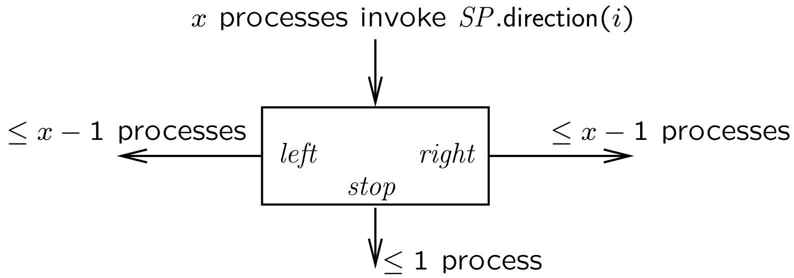
Lock-based implementation	Mutex-free implementation
	Obstruction-freedom
Deadlock-freedom	Non-blocking
Starvation-freedom	Wait-freedom

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

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

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

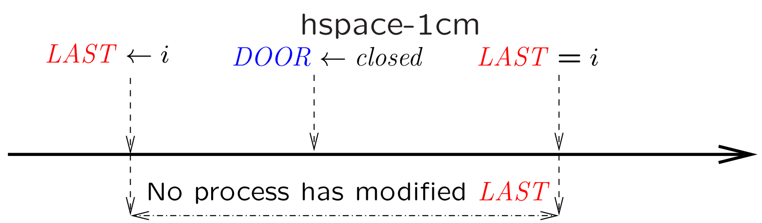


- Works in the weakest model: $\mathcal{ARW}_{n,n-1}[\emptyset]$

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

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

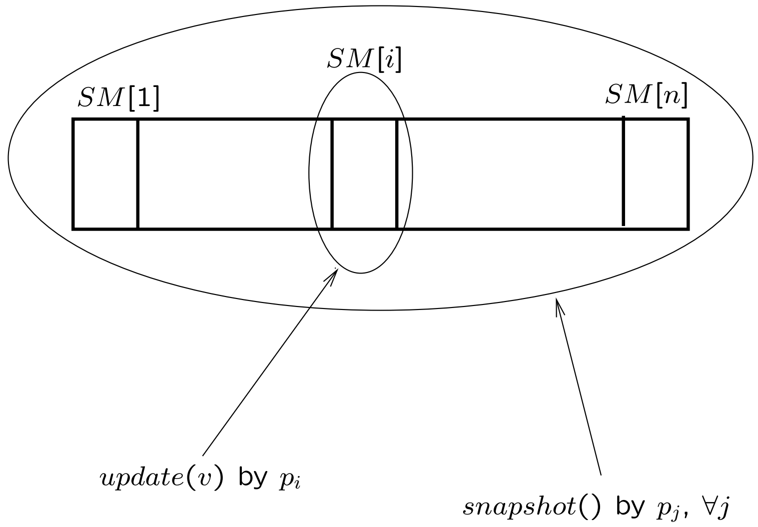

A very simple wait-free object: the Splitter (4)



A snapshot Object

- Keeps data provided by processes
- When p_i invokes store(v) it defines v as its last deposited value
- A process invokes snapshot to get the values deposited by the processes
- Everything has to appear as if the operations were executed instantaneously (at some time between their invocation and their termination)

Snapshot operations

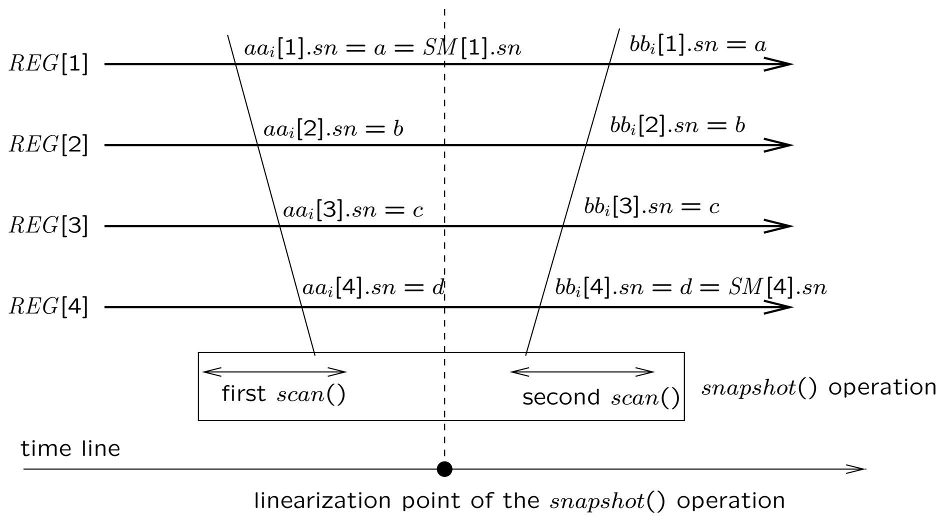


Underlying idea

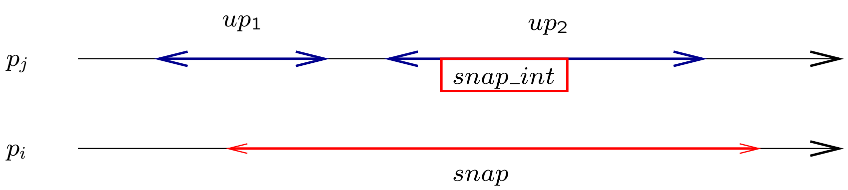
operation update (v)
    $sn_i \leftarrow sn_i + 1$; % local seq number generator %
    $SM[i] \leftarrow (v, sn_i)$ % atomic write %

operation snapshot
    while *true* **do**
      $A_i \leftarrow$ *scan*; $B_i \leftarrow$ *scan*;
      % double "asynchronous" scan %
      if $(\forall j : A_i[j].sn = B_i[j].sn)$ **then** return $(A_i.val)$ **end_if**
          % $A_i.val = [A_i[1].val, \ldots, A_i[n].val]$ %
    end_while

Snapshot: Partial proof (easy)



How an update can help a snapshot



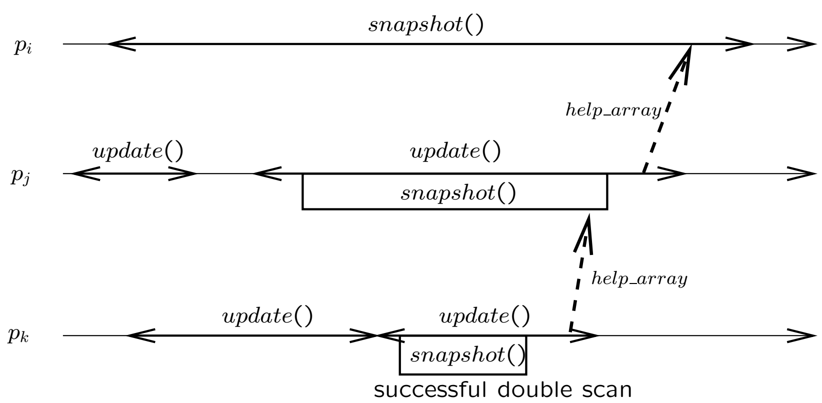
Afek et al.s algorithm (1)

operation update (v):
  $help\_array_i \leftarrow$ snapshot();
  $sn_i \leftarrow sn_i + 1$;
  $SM[i] \leftarrow (v, sn_i, help\_array_i)$

Afek et al.s algorithm (2)

operation snapshot:
  $could\_help_i \leftarrow \emptyset$;
  while $true$ **do**
    $A_i \leftarrow$ *scan*; $B_i \leftarrow$ *scan*; % double "asynch" collect %
    if $(\forall j : A_i[j].sn = B_i[j].sn)$
      then return $(A_i.val)$
      else **for** $j : 1 \leq j \leq n$ **do**
          if $(A_i[j].sn \neq B_i[j].sn)$ **then**
            if $(j \in could\_help_i)$
              then return $(B_i[j].help\_array)$
              else $could\_help_i \leftarrow could\_help_i \cup \{j\}$
          end_if end_if
        end_for
    end_if
  end_while

Snapshot: Proof



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

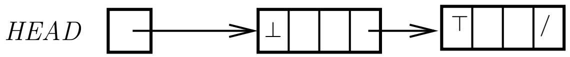
- Heller S., Herlihy M.P., Luchangco V., Moir M., Scherer W.III and Shavit N., A lazy concurrent list-based algorithm. *Parallel Processing Letters*, 17(4):411-424, 2007.

Internal representation

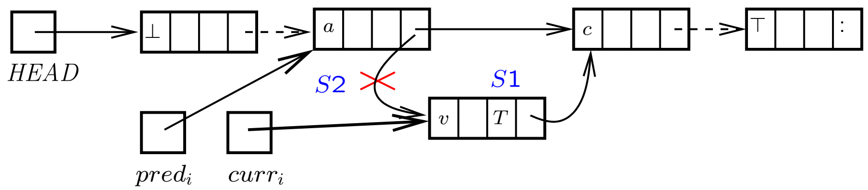
- linked list pointed to by $HEAD$
- A cell of the list (say NEW_CELL) is made up of
  - ⋆
  - ⋆ $NEW\_CELL.val$ which contains a value (element of the set).
  - ⋆ $NEW\_CELL.out$: Boolean set to $true$ when the corresponding element is suppressed from the list
  - ⋆ $NEW\_CELL.lock$: lock used to ensure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

Initial state

- The set is represented with a sorted linked list
- All operation algorithms are based on list traversal



Operation S.remove(v): behavior



Operation S.remove(v): algorithm

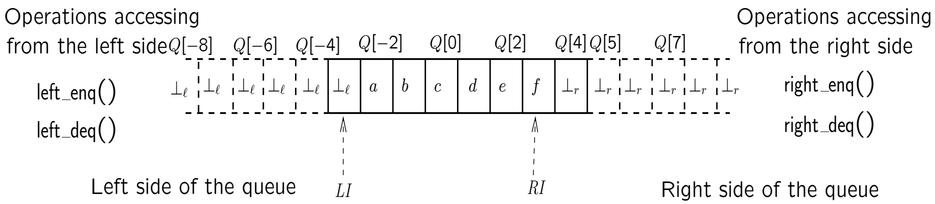
operation S.remove(v) **is**
 $pred \leftarrow HEAD$; $curr \leftarrow (HEAD\downarrow).next$;
 while $((curr\downarrow).val < v)$
  do $pred \leftarrow curr$; $curr \leftarrow (curr\downarrow).next$ **end while**;
 $((pred\downarrow).lock)$.acquire_lock(); $((curr\downarrow).lock)$.acquire_lock();
 $valid \leftarrow false$;
 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.

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**.

While the operations S.remove(v) and S.add(v) are loock-based the operation S.contain(v) is wait-free!

Example of a double-ended queue



Favorable circumstances: concurrency-free patterns

- In $\mathcal{ARW}_n$[Compare&Swap]:
  Herlihy M.P., Luchangco V. and Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23th Int'l IEEE Int'l Conference on Distributed Computing Systems (ICDCS'03)*, IEEE Press, pp. 522-529, 2003
- In $\mathcal{ARW}_n$[LL/SC]:
  Taubenfeld G., Contention-sensitive data structure and algorithms. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 157-171, 2009

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" where those are application-dependent

A non-blocking abortable (bounded) stack

- Bounded: size of the stack $= k$
- Works in $\mathcal{ARW}_{n,n-1}$[Compare&swap]
- Any operation terminates in concurrency-free pattern
- Non-blocking $\Rightarrow$ in the presence of concurrency always at least one operation invocation terminates

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;
$\boxed{old \leftarrow X}$; % read of X
Begin of a speculative execution:
any sequence of statements possibly
involving accesses to the shared memory
and computing a new value new to assign to X
End of speculative execution;
if $\boxed{X\text{.compare\&swap}(old, new)}$ % conditional write of X
then statements S1 % success : commit
else statements S2 % abort : restart
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

A non-blocking abortable bounded stack

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

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

Stack internal 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 internal 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

Abortable push: ab_push()

operation ab_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$(\langle stacktop, seqnb-1 \rangle, \langle value, seqnb \rangle)$.

On any entry of the stack:
the x-th write must follow the $(x-1)$-th

Part VII

Conclusion

What do we have visited?

- Concurrent objects
- Different types of objects
- Safety and progress conditions
- Lock-based vs mutex-free implementations
- Notion of a hybrid implementation
- Abortable objects

A few books on the topic

- Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, 2006 (ISBN 0-131-97259-6)
- Herlihy M. and Shavit N., *The art of multiprocessor programming*. Morgan Kaufmann, 508 pages, 2008 (ISBN 978-0-12-370591-4).
- Raynal M., *Concurrent programming: algorithms, principles, and foundations.* Springer, 530 pages, 2013 (ISBN 978-3-642-32026-2)

More important, HE tolds me

"Algorithms lie at the core of computing science"

And, maybe more important, SHE tolds me

"Synchronization and non-determinism are among its most fundamental concepts"